NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

ENVIRONMENT BEHAVIOR MODELS FOR REAL-TIME REACTIVE SYSTEM TESTING AUTOMATION

by

Muharrem Ugur Aksu

September 2006

Co-Advisors: Mikhail Auguston
Man-Tak Shing

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 2006	Master's Thesis

4. TITLE AND SUBTITLE Environment Behavior Models for Real-Time Reactive System Testing Automation	5. FUNDING NUMBERS
6. AUTHOR(S) Muharrem Ugur Aksu	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
Approved for public release; distribution is unlimited.	

13. ABSTRACT (maximum 200 words)

We explored the effectiveness of using attributed event grammars (AEG) based environment behavior models as a method for testing and analyzing real-time, reactive software systems. The AEG specifies possible event traces and provides a uniform approach for automatically generating and executing test cases. We have demonstrated the approach through a case study (Paderborn Shuttle System Control Software) and performed three kinds of experiments: software correctness testing, system performance analysis and study of design alternatives.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Model-based Testing, Testing Automation, Reactive and Real-time System Testing, Attributed Event Grammars (AEG), Environment Behavior Models.	163
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

ENVIRONMENT BEHAVIOR MODELS FOR REAL-TIME REACTIVE SYSTEM TESTING AUTOMATION

Muharrem U. Aksu
1st Lt., Turkish Army
B.S., Turkish Army Academy, 2000

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE
AND
MASTER OF SCIENCE IN SOFTWARE ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 2006

Author: Muharrem U. Aksu

Approved by: Mikhail Auguston
Co-Advisor

Man-Tak Shing
Co-Advisor

Peter Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

We explored the effectiveness of using attributed event grammars (AEG) based environment behavior models as a method for testing and analyzing real-time, reactive software systems. The AEG specifies possible event traces and provides a uniform approach for automatically generating and executing test cases. We have demonstrated the approach through a case study (Paderborn Shuttle System Control Software) and performed three kinds of experiments: software correctness testing, system performance analysis and study of design alternatives.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

The author wants to thank Prof. Mikhail Auguston and Prof. Man-Tak Shing for their guidance and patience during the work in performing this thesis study.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Statistics show that any nontrivial software system will have, on average, one to three errors per hundred statements no matter how hard we try to prevent those errors [1]. On the other hand, the process of testing software and finding those errors is not easy. As a matter of fact it is time and effort consuming [6] – testing consumes almost half of the labor expended to produce a working program [1].

Software testing is the systematic process of exercising software with test cases in order to find differences between the expected behavior specified by system models and the observed behavior of the implemented system [2], [3]. The goal of software testing is to maximize the number of discovered faults through designing test cases in order to increase the reliability of the system by correcting those faults [2]. “The effort put into testing seems wasted if the tests don’t reveal bugs [1].”

When we perform system testing for software, two types of testing are crucial: functional testing and performance testing [2]. We can test the functional requirements of the system under testing (SUT) to detect errors or we can test the nonfunctional requirements and additional design goals of the SUT to find differences between nonfunctional requirements and actual system performance [2], [3]. Testing functional and performance requirements of real-time, reactive systems is more complicated than the testing of other software systems.

Real-time reactive systems are those that have timing requirements on their computations and actions and whose behaviors are primarily caused by specific reactions to external events rather than being self-generated [4]. Timing requirements of such systems are specified in terms of deadlines which can be denoted by either a point in time or a time interval by which a system action must occur [4].

Timing requirements and event-driven behavior of such systems renders testing real-time reactive systems very complicated [4], [6].

Performance and timing requirements are present at a program when keeping pace with an external physical process is mandatory [5]. In this case the SUT must respond in the time constraints imposed by the external physical process, so that the response can control the physical process [5]. The difficulty is that these requirements can only be tested by evaluating the system within the context of its operating environment [6].

The event driven behavior of real-time reactive systems implies a state-based system, and testing of a state-based system requires the use of sequences of inputs [9]. Moreover, in real-time reactive systems the response sent by the SUT may affect the next sequence of inputs sent from the environment. This behavior of real-time reactive systems forces us to create test sequences that will result from applying adaptive test cases in which the input applied at a step depends upon the output sequence that has been observed [6], [7], [9].

Continuous interaction with their environment, the adaptive nature of the environment to the responses sent from real-time reactive systems and the timing constraints on both their inputs and outputs make the interaction with a human tester during the testing of real-time reactive systems often impossible since the overhead incurred in this process will render the test results meaningless [6]. “Such systems can only be tested via an automated testing environment with processing characteristics sufficiently close to the actual operating environment [6].” A common approach to achieve this is to develop two separate models - one for the system under test (SUT) and the other for the environment with which it has an interaction or which is under its control – and run them in tandem [6]. “Hence, correct modeling of the environment is as important as the correct analysis of the system requirements [6].” Thus, creating an environment model for the SUT can give us the ability to perform both functional and performance tests on real-time reactive systems.

The agenda of this thesis is to explore the effectiveness of using environment behavior models as a method for testing and analyzing real-time,

reactive software systems. We will use automatic test case scenario generation, which is based on an attributed event grammar (AEG) model, in order to define the environment of a SUT which is a real-time, reactive software system. We will explore the extent to which experiments with a SUT embedded in an environment behavior model serve as a constructive method for testing both functional and performance requirements of real-time, reactive software systems through quantitative and qualitative experiments.

THIS PAGE INTENTIONALLY LEFT BLANK

II. TASK

A. BLACK BOX TEST MODEL WITH THE USE OF AN AUTOMATED TEST GENERATOR

As we mentioned in the previous section, real-time, reactive software systems can only be tested via an automated testing environment with processing characteristics sufficiently close to the actual operating environment since the overhead incurred by using a tester will render the test results meaningless for such systems [6].

A common way of achieving this is developing two separate models - one for the system under test (SUT) and the other for the environment with which it has an interaction (or which is under its control) – and run them in tandem [6].

In this approach, the SUT is treated as a black-box and it is subjected to inputs sent from the environment, and its outputs are verified for conformance to specified behavior [1]. With the black-box model of the software to be tested, we take the user's point of view and we are only interested in the outermost functional layer of the software. Figure 1 shows the interaction between the SUT and its environment.

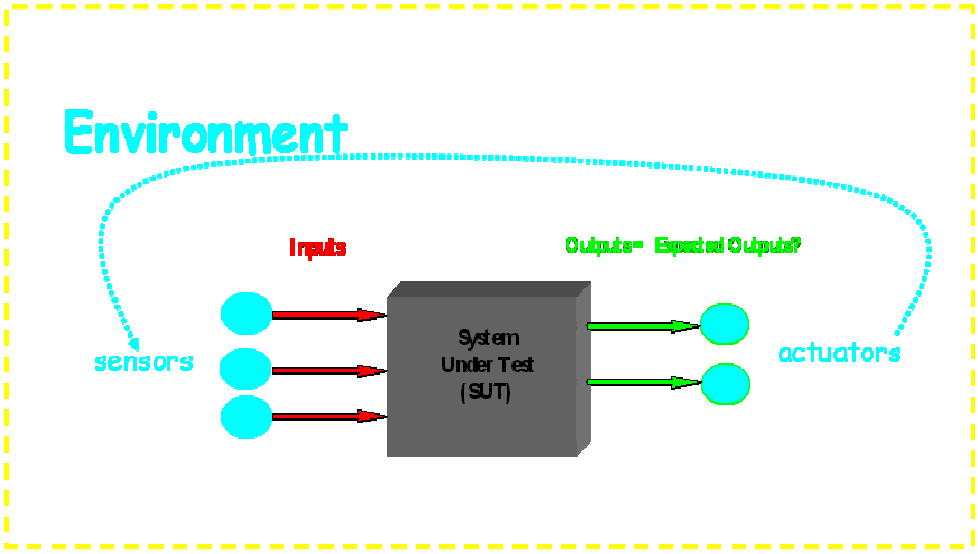

Figure 1. Black-Box Testing for Real-Time, Reactive Systems Testing (From Ref. [12]).

However, the real question is how to define the environment of a real-time, reactive software system which will send inputs to and will receive outputs from the SUT?

Our approach is to define the environment of the SUT using attributed event grammars (AEG) and to generate random event-traces which will interact with the SUT as a test driver and run them together with the help of a run time monitor. “Hence, correct modeling of the environment is as important as the correct analysis of the system requirements [6].”

The process of generating test drivers from attributed event grammar (AEG) models is achieved with the help of an automated test generator, the first version of which takes an AEG code and generates a test driver in C. The proposed overall test model with the use of the automated test generator is described in Figure 2.

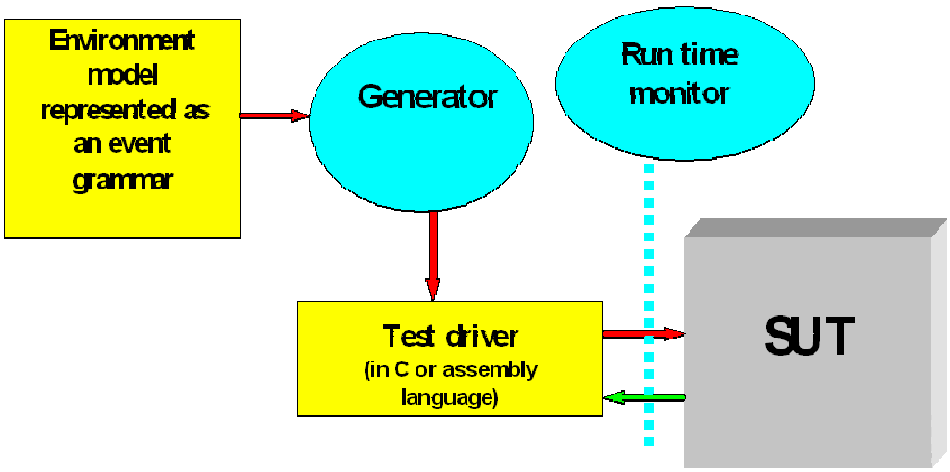

Figure 2. The Use of Automated Test Generator (From Ref. [7]).

Specifically, the automated test generator takes an AEG environment model that specifies a set of possible scenarios (or use cases) for the SUT, derives a random event trace from it according to the probabilities and iteration guards in the AEG, and generates a test deriver in C [7].

B. WHAT ARE ATTRIBUTED EVENT GRAMMARS (AEG)?

Attributed Event Grammars (AEG) [6], [7] are based on the notion of an **event,** which is any detectable action in the environment that could be relevant to the operation of the SUT. A keyboard button pressed by the user, a group of alarm sensors triggered by an intruder, a particular stage of a chemical reaction monitored by the system, and the detection of an enemy missile are all examples of events. An event is usually a time interval, and has a beginning, an end, and duration. An event has **attributes**, such as type and timing attributes.

Two basic relations are defined for events: **precedence** (PRECEDES) and **inclusion** (IN). Two events may be ordered in time, or one event may appear inside another event. The behavior of the environment can be represented as a set of events with these two basic relations defined for them (**event trace**). The basic relations define a partial order of events. Two events are not necessarily ordered, that is, they can happen concurrently. Usually event traces have a specific structure (or constraints) in a given environment.

The structure of possible event traces can be specified by an **event grammar**. Here identifiers stand for event types, sequence denotes precedence of events, (…|…) denotes alternative, (*…*) means repetition zero or more times of ordered events, […] denotes an optional element, {a, b} denotes a set of two events a and b without an ordering relation between them, and {*…*} denotes a set of zero or more events without an ordering relation between them.

1. Attributed Event Grammar Axioms

The rule A: B C means that an event of the type A contains (IN relation) ordered events of types B and C, correspondingly (PRECEDES relation). Both relations imply partial order and are transitive, noncommutative, nonreflexive, and satisfy distributivity constraints. The following axioms should hold for any event trace where a, b, c and d are events of any type:

Mutual Exclusion of Relations

Axiom 1.1: a PRECEDES b $\Rightarrow$ $\neg$ (a IN b)

Axiom 1.2: a IN b $\Rightarrow$ $\neg$ (a PRECEDES b)

Non-commutativity

Axiom 2.1: a PRECEDES b $\Rightarrow$ $\neg$ (b PRECEDES a)

Axiom 2.2: a IN b $\Rightarrow$ $\neg$ (b IN a)

Transitivity

Axiom 3.1: (a PRECEDES b) $\wedge$ (b PRECEDES c) $\Rightarrow$ (a PRECEDES c)

Axiom 3.2: (a IN b) $\wedge$ (b IN c) $\Rightarrow$ (a IN c)

Distributivity

Axiom 4.1: (a IN b) $\wedge$ (b PRECEDES c) $\Rightarrow$ (a PRECEDES c)

Axiom 4.2: (a PRECEDES b) $\wedge$ (c IN b) $\Rightarrow$ (a PRECEDES c)

Axiom 4.3: ($\forall$ a IN b ($\forall$ c IN d (a PRECEDES c))) $\Rightarrow$ (b PRECEDES d)

Figure 3. AEG Axioms (From Ref. [6]).

2. Automated Random Event-Trace Generation

Attributed event grammars (AEG) are intended to be used as a vehicle for automated random event-trace generation. It is assumed that the AEG is traversed top-down and left-to-right and only once to produce a particular event trace. Randomized decisions about what alternative to take and how many times to perform an iteration should be made during the trace generation. The major difference with traditional attributed context-free grammars is in the nature of objects defined by the grammar: instead of sequences of symbols, AEG deals with event traces, sets with two basic relations, or directed acyclic graphs.

The event grammar defines a set of possible event traces – a model of behavior for a certain environment. The purpose is to use it as a production grammar for random event trace generation by traversing grammar rules and making random selections of alternatives and numbers of repetitions. All generated concurrent events within sets start simultaneously.

Each event type may have a different attribute set. An event grammar can contain attribute evaluation rules similar to the traditional attribute grammar. Attribute values are evaluated during the AEG traversal. The */action/* is performed immediately after the preceding event is completed.

3. An Attributed Event Grammar Example

The interface with the SUT can be specified by an action that sends input values to the SUT. This may be a subroutine in a common programming language like C that hides the necessary wrapping code. In the following example of specifying a variety of use case scenarios for a simple calculator, we suppose that the SUT should receive a message about the button pressed by the user corresponding to the appropriate wrapper subroutine shown in Figure 4: **enter_digit()**, **enter_operation()**, and **show_result()**.

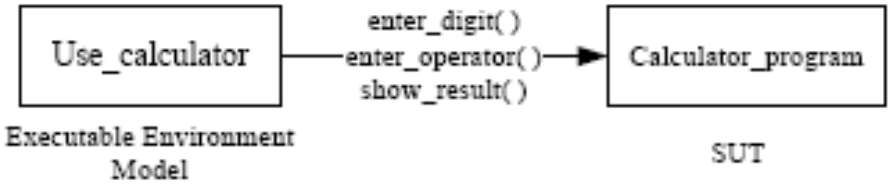

Figure 4. The Environment Model for the Calculator Scenario (From Ref. [6])

Some event types in this model have attributes associated with them.

```
Perform_calculation  result
Enter_number         digit, value
Enter_operator       operation

Use_calculator: (* Perform_calculation *);
Perform_calculation:
        Enter_number Enter_operator Enter_number
        WHEN (Enter_operator.operation == '+')
                / Perform_calculation.result =
                        Enter_number[1].value +
                        Enter_number[2].value; /
        ELSE
                / Perform_calculation.result =
                        Enter_number[1].value −
                        Enter_number[2].value; /
[P(0.7) Show_result];
```

Figure 5. The Attributed Event Grammar (AEG) Model for the Calculator Scenario (From Ref. [6])

The **WHEN** clause provides for conditional action, **Enter_number[1]** refers to the first occurrence of an event in the rule Perform_calculation, and correspondingly, **Enter_number[2]** refers to the second occurrence. In this example all event attribute evaluation can be accomplished at the generation time. The optional clause **Show_result** will be generated according to the probability P(0.7) assigned to it. The value of attribute **Perform_calculation.result** can be used as a test oracle for this particular part of the test case.

```
Enter_number:
        / Enter_number.value= 0; /
        (* Press_digit_button
                / Enter_number.digit = RAND[0..9];
                  Enter_number.value =
                   Enter_number.value * 10 +
                        Enter_number.digit;
                enter_digit(Enter_number.digit); / *) (1..6);

Enter_operator:
        ( P(0.5) / enter_operation('+');
                Enter_operator .operation= '+'; / |
        P(0.5) / enter_operation('-');
                Enter_operator .operation= '-'; / ) ;

Show_result: /show_result();/ ;
```

Figure 6. The Attributed Event Grammar (AEG) Model for the Calculator Scenario (From Ref. [6])

The action **/Enter_number.digit = RAND[0..9];/** assigns a random value from the interval 0..9 to the **digit** attribute. Each time the rule for Enter_number event is traversed, the number of iterations will be selected at random from interval 1..6. The traversal of AEG rules is performed top-down and from left to right, and for each iteration the attributes **Enter_number.digit** and **Enter_number.value** are recalculated. The action **enter_digit(Enter_number.digit)** feeds the corresponding value to the SUT.

When traversing this rule, the choice of action sending the operator symbol to the SUT is made based on the probability P(prob) assigned to the corresponding alternative. The event **Show_result**, when generated, will trigger a call to the wrapper subroutine that sends a message to the SUT.

We can generate a large number of **Use_calculator** scenarios (event traces) satisfying this AEG and each event trace will satisfy the constraints imposed by the event grammar. The event trace generated from the AEG traversal contains both events and actions that should be performed at corresponding time moments. The actions (wrapper subroutine calls in this example) can be extracted from the event trace and assembled into test-driver code which will perform those actions according to the timing attributes calculated during the trace generation. Thus, the event trace is used as a "scaffold" for test driver generation. Separation of the generation phase from test execution is essential for the performance of the generated test driver: event selection and attribute evaluation can be performed at generation time, with test drivers containing only wrapper calls to interact with the SUT, that is, the "scaffolding" is removed.

C. AUTOMATED TEST GENERATION

As explained in the previous sections, the automated test generator takes an attributed event grammar (AEG) environment model that specifies a set of possible scenarios (or use cases) for the system under testing (SUT), sorts (can be sorted according to the timing attributes) and derives a random event trace from it according to the probabilities and iteration guards in the AEG, and generates a test driver that will feed the SUT with inputs and will capture SUT outputs [7]. Please refer to Figure 2 for more detail.

The underlying principles of the automated test generator [7] are as follows:

a. Parallel event threads (for sets, like {A, B}) are implemented by interleaving events/actions within them.

b. All loops in an AEG model are unfolded either using explicit iteration guards, or by assuming a random number of iterations. Recursion is not supported in the current version.

c. Attributes are evaluated mostly at the generation time, but those dependent on SUT outputs (on catch clauses) are postponed till the run time. Certain parts of generated event trace may depend on those attribute values (for instance, because the delayed attribute participated in the when clause), in this case both alternatives for the expected trace segment are generated but protected by boolean flags, so that at the test run time only the alternatives for which the guard is enabled will be executed.

D. PADERBORN SHUTTLE SYSTEM

In the context of a series of new research projects at the University of Paderborn a new rail-based transport system is being developed – Paderborn Shuttle System [10] – and the requirements defined below are a slightly modified version of this case study. The system is intended to enable individual transport of people, which today is mainly conducted by cars and trucks, by autonomously acting shuttles on rail. This autonomy is supposed to eliminate the disadvantages of modern trains concerning individual transport.

1. System Overview

The following simplified system is the basis of the case study. Consider a railway. The railway consists of interconnected stations. Shuttles bid for orders to transport passengers between certain stations. Successful completion of an order results in a monetary reward for the shuttle involved. New orders are made known to all shuttles, thus all shuttles can make an offer. The shuttle with the best, i.e. lowest, offer will receive the assignment. Using the tracks will incur a toll, depending on the distance covered. Maintenance of the shuttles is possible at any station and will cost both time and money.

2. The Railway Network And Stations

The railway network consists of stations and tracks between stations. Tracks can be traveled in both directions. Railway Network is represented by a directed graph as shown in the picture below. Each track at the railway network has the same distance weight.

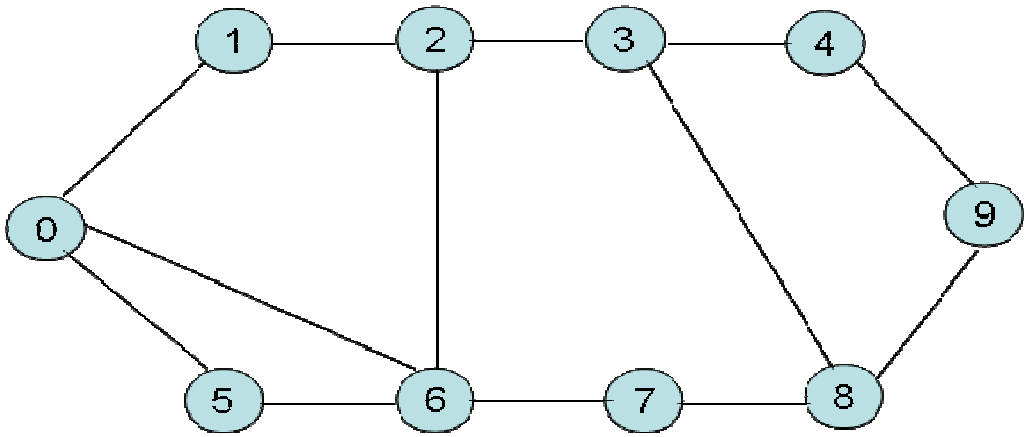

Figure 7. Proposed Paderborn Shuttle System Railway Network.

Any number of shuttles can be present at a station at the same time. The duration of a shuttle's stay at a station is not considered maintenance time. Maintenance must be explicitly scheduled.

3. Orders

Orders are made known to all shuttles by a broker. An order defines start and destination stations and the distance between those stations. Order assignment follows a strict pattern. First, all shuttles are informed of the new order and any shuttle can make an offer, which defines the payment it will receive after successful completion of that order. Shuttles calculate their bids according to the distance of each order informed by the broker. The shuttle having made the lowest offer will receive the assignment. In the event of two equal offers, the assignment will go to the shuttle that first made the offer.

4. Shuttles

Order processing is handled by the shuttles. Every shuttle can transport passengers up to a maximum capacity determined at the start of the simulation. This means that a shuttle can transport more than one passenger who requests the same order or a subset of the order for which the shuttle has made a bid as long as the number of passengers does not exceed the maximum capacity. To complete an order a shuttle has to travel to the start station, load the order and then proceed to the destination station to unload. Order-processing begins with the loading at the start station and ends with unloading at the destination. Loading or unloading at other stations is permitted to load passengers requesting a subset of the currently processed order. A travel decision is made by the broker before sending an order to the shuttles and at each station during the journey shuttles request the next station to travel in order to complete their orders.

5. Income and Expenses

At the beginning, every shuttle will receive a fixed starting capital. Afterwards, a shuttle's only means of income generation is to successfully complete orders. Payment occurs after an order is delivered. If a shuttle is at a station and its account shows a negative balance, it will not be permitted to leave this station, and is retired.

There are the following three different types of costs:

a. Toll:

Traveling from station to station costs a fee and this fee is the same for each track.

b. Maintenance:

After traveling a certain distance, maintenance has to be carried out.

The distance depends on the number of tracks. If a shuttle exceeds this limit, maintenance will be carried out at the next station automatically, and it

will not be able to leave the station until maintenance is finished. Payment is immediate.

c. Penalties:

If an order has not been delivered in time, a penalty will be imposed. If a shuttle currently carries the order, it has to complete it.

E. AEG ENVIRONMENT MODEL FOR PADERBORN SHUTTLE SYSTEM

The following basic SHUTTLE SYSTEM environment model has been used as a starting point to create several variations of environment models for SHUTTLE SYSTEM testing purposes, in particular by selecting different probabilities P(prob) for customer arrival rate (customer order request frequency), choosing random values for number of shuttles in the Shuttle System and manipulating shuttle payment and wear values dynamically at run time.

The environment model specification in AEG starts with declarations of global parameters.

```
GLOBAL {
    int transit_fee; /* toll for using the tracks */
    int transit_wear; /* wear incurred for using the tracks */
    int maintenance_wear; /* restored wear value after maintenance */
    int maintenance_fee; /* maintenance fee */
}
```

Figure 8. Paderborn Shuttle System AEG Model: Global Variables of ShuttleSystem Defined

Attributes for each rule are defined in corresponding declaration sections.

```
RULE Shuttle {
    int start; /* start station of an order */
    int destination; /* final station of an order */
    int shuttle_id; /* unique shuttle identification no */
    int shuttle_at_station; /* current location of a shuttle */
    int capital; /* capital status of a shuttle */
    int wear; /* maintenance status of a shuttle */
    int retired; /* a flag for shuttle bankruptcy */
    int payment; /* money received after order completion */
    int bid; /* bid made by a shuttle for a given order */
    int ord_confirmed; /* a flag for order assignment  */
    int received_order; /* a flag for order offers from broker */
    int distance; /* number of stations for an order */
    int order_request_no; /* auxiliary variable */
}

RULE Customers {
    int requested_start_station;
    int requested_destination_station;
}
```

Figure 9. Paderborn Shuttle System AEG Model: Top Level Rules; Shuttle & Customers Defined with Their Event Attributes

Global parameters are initialized at the top level rule ShuttleSystem. Initial settings for global parameters and attributes are obtained by calling auxiliary subroutines for the convenience of changing them dynamically during a long series of test runs [7].

The behavior model for ShuttleSystem is represented by two concurrent threads of events: Shuttles and Customers.

```
ShuttleSystem :
    /
    transit_fee = get_transit_fee();
    transit_wear = get_transit_wear();
    maintenance_wear = get_maintenance_wear();
    maintenance_fee = get_maintenace_fee();
    /
    {Shuttles, Customers};
```

Figure 10. Paderborn Shuttle System AEG Model: Behavior of ShuttleSystem Defined & Global Variables Initialized

The behavior of Shuttles is represented by a number of concurrent Shuttle Events.

```
Shuttles:
    /***CHANGE NUMBER OF SHUTTLES HERE***/
    {*Shuttle*}(==5);
```

Figure 11. Paderborn Shuttle System AEG Model: Event Shuttles Defined with Concurrent Shuttle Events

The behavior of Customers is represented by sending random requests with a probability P(prob) to the MANAGER. The (EVERY 10 sec) clause guides the event trace generation with the desired time stamps [7]. Using a combination of a period of time (with the use (EVERY 10 sec) statement) and a probability enables us to simulate the aperiodic nature of customer arrivals (customer order request events). The (==1500) construct determines number of iterations generated, that is, the duration of the customer request events will be approximately 250 minutes [7].

```
Customers:
    /
    Customers.requested_start_station = 0;
    Customers.requested_destination_station = 0;
    /
    (*  [P(70)/get_random_request(Customers.requested_start_station,
                                  Customers.requested_destination_station);
            send_customer_request(Customers.requested_start_station,
                                  Customers.requested_destination_station);/]
        /***CHANGE NUMBER AND FREQUENCY OF CUSTOMER REQUESTS HERE***/
    *)(==1500)(EVERY 10 sec);
```

Figure 12. Paderborn Shuttle System AEG Model: Event Customers Defined

The next AEG model defines the behavior of each Shuttle. First Shuttle attributes are initialized. The Shuttle.shuttle_id attribute is initialized by calling an auxiliary subroutine which returns a unique value that is incremented by one

each time it is called. This will help us to define a unique id for each Shuttle thread in the ShuttleSystem event. Initial settings for Shuttle capital values and starting stations for each are also obtained by calling auxiliary subroutines for the convenience of manipulating those values dynamically.

```
Shuttle :
    /
    Shuttle.shuttle_id = unique_id();
    Shuttle.start = 0;
    Shuttle.destination = 0;
    Shuttle.shuttle_at_station = get_shuttle_at_station();
    Shuttle.capital = get_capital();
    Shuttle.wear = maintenance_wear;
    Shuttle.retired = 0;
    Shuttle.payment = 0;
    Shuttle.bid = 0;
    Shuttle.ord_confirmed = 0;
    Shuttle.received_order = 0;
    Shuttle.distance = -1;
    Shuttle.order_request_no = 0;
```

Figure 13. Paderborn Shuttle System AEG Model: Attributes of Event Shuttle Initialized

Shuttle behavior is defined with the following sequential and iterated process. Shuttles send ready messages only once at the beginning of the process to inform the Manager of their existence and their being ready. Then Shuttles request orders, receive orders sent by the Manager and send their bids for each order to the Manager. If a confirmation message is received from the Manager they start processing the order confirmed. If the current station of a Shuttle is not the same as the start station of the order, Shuttles request and move to the next station until they are at the start station of the order. Then they start requesting and moving to the next station until they reach the destination of the order. When they are at the destination they inform the Manager of successful completion of the order by sending an order completed message.

The main loop encapsulates the overall behavior of requesting orders, sending bids, receiving order confirmations and processing confirmed orders and this behavior is iterated 50 times for each shuttle in the system.

The loops are guarded by constant values which determine the maximum number of iterations for a given set of sequential events. WHEN construct is used to break out of a loop if the given conditional guard is satisfied. For instance the move behavior of a Shuttle is implemented by a loop of 5 iterations which is the maximum distance between two farmost stations (the longest distance to be traversed by shuttles at the worst case) and this loop is iterated until a given Shuttle reaches its destination station in order to complete its order.

The attribute evaluation statements and WHEN constructs make the event generation dependent on the previous events in the trace [7].

```
send_ready(Shuttle.shuttle_id);/
(*
    /Shuttle.order_request_no = 0;/
    (*
        /request_order(Shuttle.shuttle_id, Shuttle.order_request_no);/
        wait_order_and_send_bid
          /***WHEN THERE IS ONLY ONE MORE ORDER TO BE OFFERED IN THE QUEUE…***/
          /***REQUEST FOR AN ORDER ONE LAST TIME AND WAIT FOR ORDER CONF.***/
        WHEN (Shuttle.order_request_no == -1)
        (
            /request_order(Shuttle.shuttle_id, Shuttle.order_request_no);
            BREAK; /
        )
        /***WHEN THERE IS NO AVAILABLE ORDER IN THE QUEUE…***/
        /***WAIT ONE ORDER PROCESSING PERIOD OF TIME AND REQUEST AGAIN***/
        WHEN (Shuttle.order_request_no == -2)
            /BREAK;/
          /***MAKE SURE THIS NUMBER IS EQUAL TO NUMBER OF SHUTTLES***/
    *)(==5)
    wait_order_confirmation
    WHEN (Shuttle.ord_confirmed)
    (
        /Shuttle.payment = Shuttle.bid;/
        (*
            WHEN (ENCLOSING Shuttle.shuttle_at_station != ENCLOSING Shuttle.start)
            (
                /move_to_start_station(Shuttle.shuttle_id, Shuttle.shuttle_at_station);/
                wait_next_station
                process_move
            )
            ELSE /BREAK;/
        /***THIS IS THE MAX DISTANCE BETWEEN TWO FARMOST STATIONS***/
        *)(==5)
        (*
            WHEN (ENCLOSING Shuttle.shuttle_at_station != ENCLOSING Shuttle.destination)
            (
                /request_next_station(Shuttle.shuttle_id, Shuttle.shuttle_at_station);/
                wait_next_station
                process_move
            )
            ELSE /BREAK;/
        /***THIS IS THE MAX DISTANCE BETWEEN TWO FARMOST STATIONS***/
        *)(==5)
        process_order_completion
    )
/***MAIN LOOP - INCREASE TO GENERATE MORE DATA***/
*)(==50);
```

Figure 14. Paderborn Shuttle System AEG Model: Behavior of Event Shuttle Defined

A sequence of events can be grouped under another event in order to increase the readability of a complex event. It is also useful to group a sequence of events and create a new event encapsulating the overall behavior of those events when they are used repeatedly in the attributed event grammar (AEG) model. This approach can save space and decrease the total number of lines of AEG codes.

The next two events define the behavior of receiving an order and sending bids for those orders to the Manager. The CATCH construct represents the external event of receiving a message from the MANAGER [7] and it is implemented as a function call order(ENCLOSING Shuttle.shuttle_id, ENCLOSING Shuttle.start, ENCLOSING Shuttle.destination, ENCLOSING Shuttle.distance, ENCLOSING Shuttle.order_request_no) which returns a True value when MANAGER has issued corresponding output. The WAIT behavior for receiving a message from the MANAGER is represented by a CATCH clause that is repeated more than once inside a loop. This is very useful in simulating the time interval in which Shuttles are expecting a message from the MANAGER. This might be a timing constraint for the SUT and the overall system function may fail if this constraint is not satisfied or the event stream may proceed to the next action if there is no input from the SUT at that time interval; the latter is the case for our model. The WAIT and CATCH constructs encapsulate the interface with the OMNeT++ message queue [7].

The construct ENCLOSING Shuttle provides for the event wait_order_and_send_bid to access the attributes of the parent event Shuttle which are not within the scope of this rule. This reference mechanism is convenient for event attribute propagation over the generated event trace [7].

Shuttles WAIT for 10 seconds to receive an order and calculate and send their bids if an order is received. Calculating and sending bids are defined as another event for simplicity and readability reasons.

```
wait_order_and_send_bid:
    (* CATCH order(ENCLOSING Shuttle.shuttle_id, ENCLOSING Shuttle.start,
                    ENCLOSING Shuttle.destination, ENCLOSING Shuttle.distance,
                    ENCLOSING Shuttle.order_request_no)
        /ENCLOSING Shuttle.received_order = 1;/
        calculate_and_send_bid
      END_CATCH
    *)(==2)(EVERY 5 sec);

calculate_and_send_bid:
    WHEN(ENCLOSING Shuttle.received_order)
    (
        /ENCLOSING Shuttle.bid = calculate_bid(ENCLOSING Shuttle.distance);/
        WHEN (ENCLOSING Shuttle.order_request_no != -2)
            /send_bid(ENCLOSING Shuttle.shuttle_id, ENCLOSING Shuttle.bid,
                    ENCLOSING Shuttle.start, ENCLOSING Shuttle.destination);/
        /ENCLOSING Shuttle.received_order = 0;/
    );
```

Figure 15. Paderborn Shuttle System AEG Model: Sub-events of Event Shuttle Defined

Shuttles WAIT for 10 seconds to receive an order confirmation and continue with the next action if an order confirmation is not received. The WAIT function encapsulates the interface with the OMNeT++ message queue [7].

```
wait_order_confirmation:
    (* CATCH order_confirmed(ENCLOSING Shuttle.shuttle_id,
                            ENCLOSING Shuttle.ord_confirmed,
                            ENCLOSING Shuttle.start,
                            ENCLOSING Shuttle.destination,
                            ENCLOSING Shuttle.bid)
      END_CATCH
    *)(==2)(EVERY 5 sec);
```

Figure 16. Paderborn Shuttle System AEG Model: Sub-events of Event Shuttle Defined

Shuttles WAIT for 20 seconds to receive next station values and this time interval simulates the time elapsed in order to move from one station to the next. The WAIT function encapsulates the interface with the OMNeT++ message queue [7].

```
wait_next_station:
    (* CATCH next_station(ENCLOSING Shuttle.shuttle_id,
                          ENCLOSING Shuttle.shuttle_at_station)
      END_CATCH
    *)(==2)(EVERY 10 sec);
```

Figure 17. Paderborn Shuttle System AEG Model: Sub-events of Event Shuttle Defined

The event process_move manipulates the capital, wear and maintenance attributes of Shuttles for each move from one station to the next.

```
process_move:
    WHEN (ENCLOSING Shuttle.wear > 0)
    (
        /ENCLOSING Shuttle.capital =
            ENCLOSING Shuttle.capital - transit_fee;
         ENCLOSING Shuttle.wear =
            ENCLOSING Shuttle.wear - transit_wear; /
    )
    ELSE
    (
        /ENCLOSING Shuttle.capital = ENCLOSING Shuttle.capital –
                                     maintenance_fee - transit_fee;
        ENCLOSING Shuttle.wear = maintenance_wear;/
    );
```

Figure 18. Paderborn Shuttle System AEG Model: Sub-events of Event Shuttle Defined

The event process_order_completion manipulates the capital attributes of Shuttles by making a deposit to the capitals of Shuttles for the successful completion of an order. Upon the completion of processing an order, Shuttles send order completed messages to the MANAGER along with their capital values at that moment.

```
process_order_completion:
    [P(80) order_completed_in_time]
    [P(20) late_order_completion
        /ENCLOSING Shuttle.capital =
         ENCLOSING Shuttle.capital – get_punishment();/]
    /ENCLOSING Shuttle.capital =
            ENCLOSING Shuttle.capital + ENCLOSING Shuttle.payment;
    ENCLOSING Shuttle.ord_confirmed = 0;/
    WHEN (ENCLOSING Shuttle.capital <= 0)
    (
        /ENCLOSING Shuttle.retired = 1;/
    )
    /send_order_completed(ENCLOSING Shuttle.shuttle_id,
                          ENCLOSING Shuttle.retired,
                          ENCLOSING Shuttle.capital);/;
```

Figure 19. Paderborn Shuttle System AEG Model: Sub-events of Event Shuttle Defined

F. THE OMNET++ MODEL

1. What is OMNeT++?

OMNeT++, which stands for Objective Modular Network Testbed in C++, is an object-oriented modular discrete event network simulator primarily designed for the simulation of communication protocols, communication networks and traffic models, models of multiprocessor and distributed systems and evaluating performance aspects of complex software systems [7], [11].

An OMNeT++ model consists of hierarchically nested modules where the depth of module nesting is not limited [11]. The atomic modules are called simple modules. They are coded in C++, encapsulate the behavior and are executed as coroutines on top of the OMNeT++ simulaton kernel [7], [11].

Modules communicate with each other via message passing through gates and connections [7], [11]. Gates are the input and output interfaces of the modules and messages are sent out through output gates of the sending module and arrive through input gates of the receiving module [7]. Input and output gates are linked together via connections which represent the communication channels and can be assigned properties such as propagation delay, bit error rate and data rate [7].

OMNeT++ supports a variety of user interfaces for debugging, demonstration and batch execution purposes [11]. Advanced user interfaces allow control over simulation execution. Users can inspect the variables and messages in each module and change the values of variables during run-time. This is a very useful feature for the development and debugging phase of the model [11]. Moreover, user interfaces also demonstrate how the model works [11].

2. Paderborn Shuttle System Model in OMNeT++

Figure 22 shows the top level of the OMNeT++ simulation model for the Paderborn Shuttle System test environment. It consists of two modules, Environment - that encapsulates shuttles and customers - and Manager.

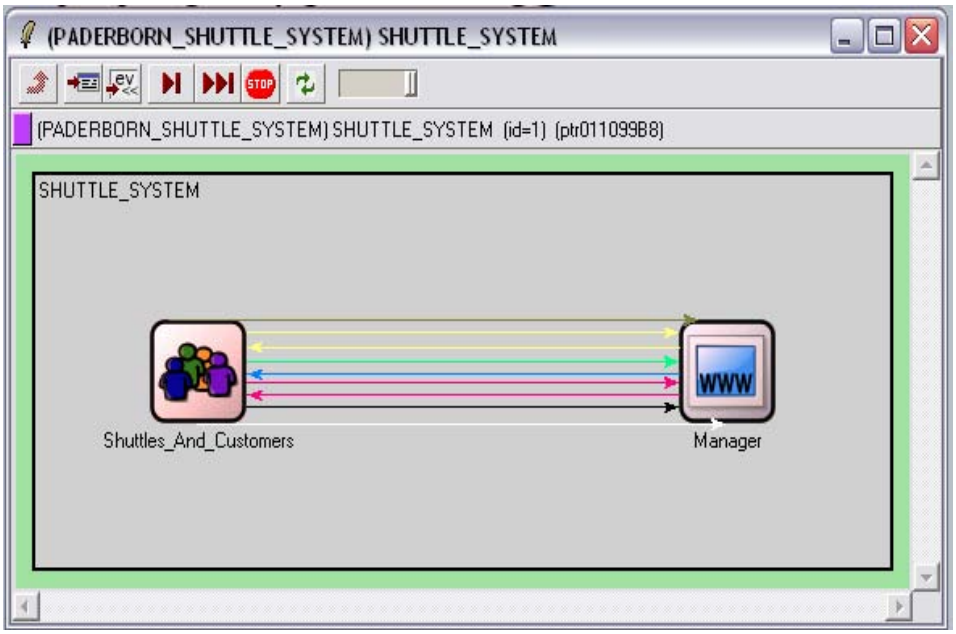

Figure 20. OMNeT++ Model for Paderborn Shuttle System.

The Shuttles_And_Customers module contains the C++ code of the test driver generated from the AEG environment model. The Manager module contains the C++ code that simulates the functional behavior of the Paderborn Shuttle System software that receives travel requests from Customers, accepts

bids from Shuttles for orders, assigns orders to Shuttles and controls capital and maintenance status of shuttles.

The messages and their parameters sent on each message channels are displayed in the figure below.

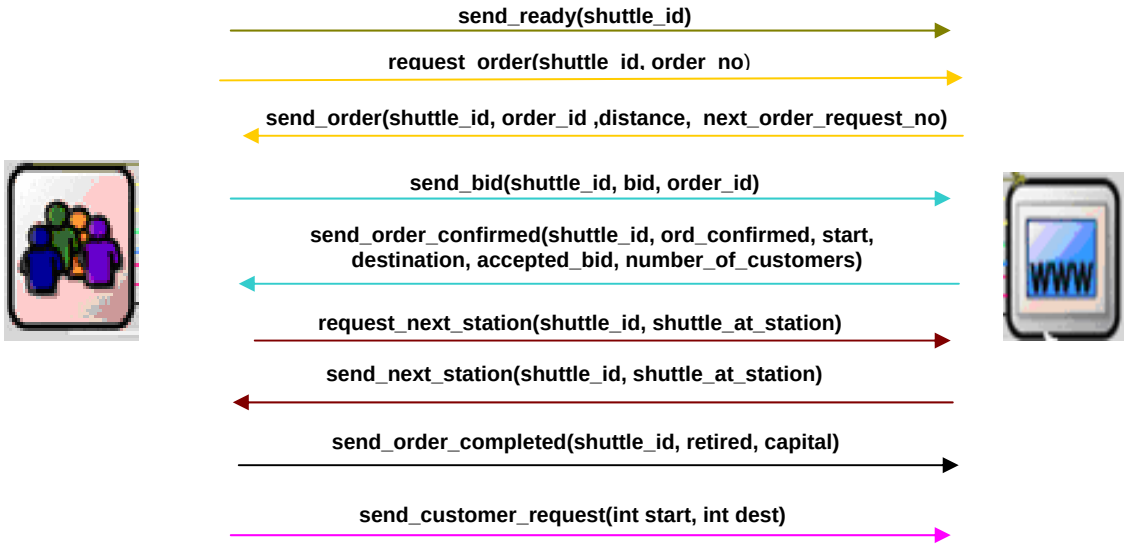

Figure 21. OMNeT++ Model for Paderborn Shuttle System That Shows the Connection Between Modules and the Messages Sent Over Those Connections.

THIS PAGE INTENTIONALLY LEFT BLANK

III. EXPERIMENTS

We conducted three types of experiments to investigate the effectiveness of the AEG-based test automation in support of correctness testing, system performance assessment and evaluation of design alternatives for the MANAGER software of the Paderborn Shuttle System.

Software correctness testing involves observing unexpected or unsafe behavior of the SUT, especially when the SUT is subjected to a series of extreme case test scenarios generated from attributed event grammar (AEG) environment models. “This type of analysis is especially useful for eliminating errors in the control software [8].”

On the other hand, system performance assessment is based on the idea of running a large number of test scenarios and gathering relevant statistical data that may give insight into the effectiveness of the SUT with respect to environmental variables [8]. From such results we can better understand which factors lead to failure in the performance of the SUT and in what way [8].

Last but not least, environment models can be very useful to support the study of design alternatives to the SUT, especially in order to measure the efficiency of different algorithm alternatives in the SUT [7]. This kind of experiment might prove very useful to test whether an algorithm that we think is efficient is truly efficient in the context of its operating environment. We can perform such experiments by subjecting each algorithm to the same scenario batches and comparing the statistical data gathered from those runs for each algorithm.

A. SOFTWARE CORRECTNESS TESTING

1. Experiment One

After analyzing the visual statistics gathered from our experiment run with the AEG based automatically generated test driver with the environmental

parameters shown in Table 1, we have identified a serious error within the MANAGER module.

Test Driver No	Number of Shuttles	Number of Order Iterations by Shuttles	Customer Arrival Rate
1	4	25	Every 10 sec. with P(70)

Table 1. Parameters of AEG Based Test Driver for Experiment One

The figure below demonstrates the number of shuttles alive (not retired), the number of shuttles processing an order (busy shuttles) and the total number of order requests waiting to be processed by shuttles in queue.

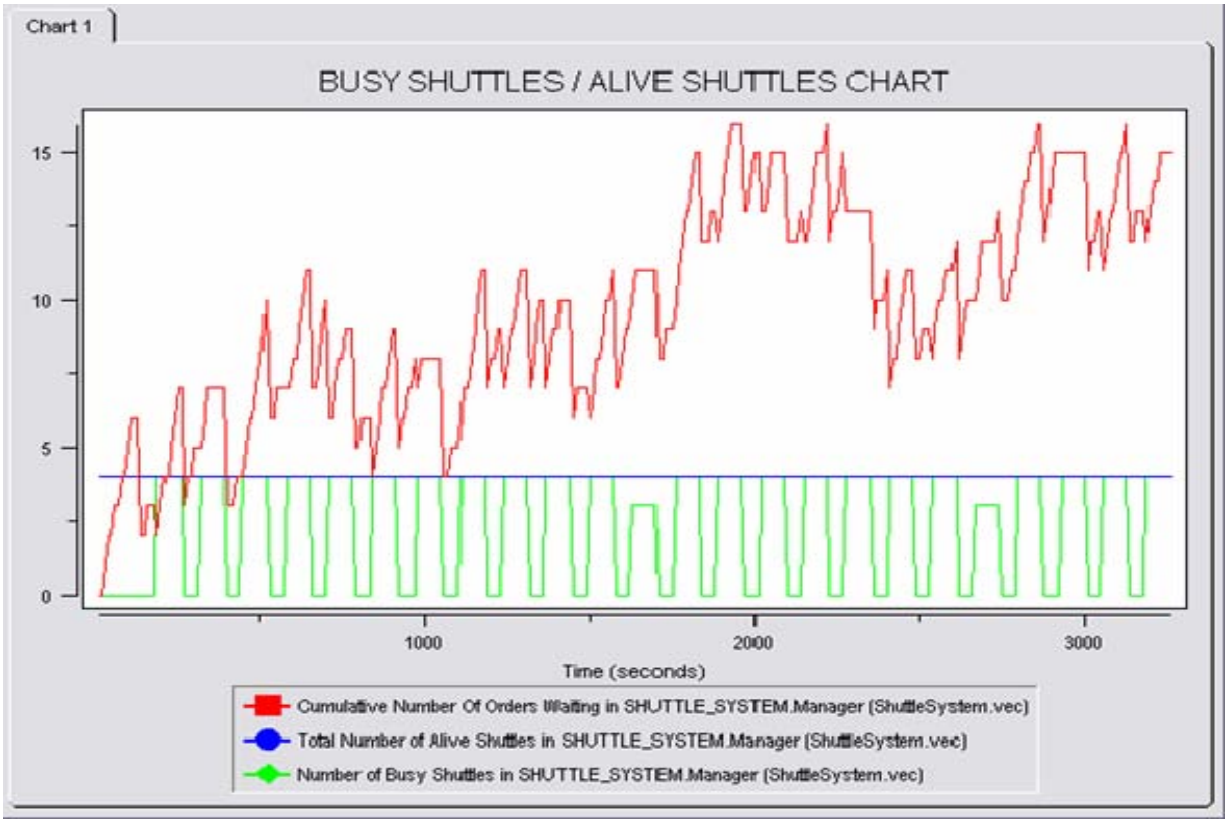

Figure 22. Visual Statistical Result of Experiment 1 (Busy Shuttles / Alive Shuttles)

The expected behavior of the experiment was to have all shuttles busy as long as the shuttles are not retired and there are orders waiting to be processed

in the queue (orders unassigned to any shuttles). However, we observed that some of the shuttles were not assigned any orders even though there were orders waiting to be processed. We traced the cause of this problem to a sub-routine (**move_unique_orders()**) in the manager module.

The Manager software (SUT) uses a sub-routine (**move_unique_orders()**) that selects unique orders from a queue (**ready_list**) that holds received customer order requests and moves those orders to another queue (**processing_list**) the orders in which will be offered to shuttles requesting orders.

A unique order is the one that where the path (a sequence of stations on a shuttles route in order to complete its order assignment) of an order in the ready list neither matches with the path of any primary orders which are in the processing list nor is a subset path of any of them.

The intended algorithm is described in the figure below.

```
MOVE_UNIQUE_ORDERS:
        for(each order A in processing_list) {
          for(each order B in ready_list) {
              if(B is NOT on the same route with A)
                  then move B to processing_list
              if(processing_list.SIZE EQUALS TO number of shuttles alive)
                                  break;
      }
  }
```

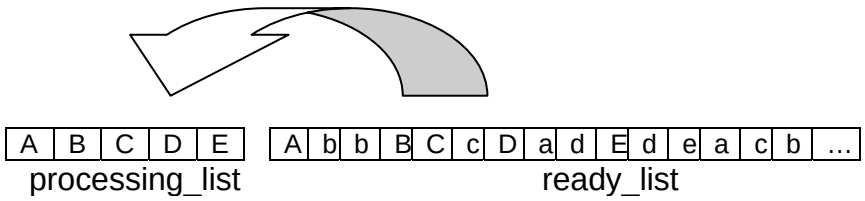

Figure 23. Pseudo Code for **move_unique_orders()** Sub-routine

After carefully analyzing our algorithm we realized that it identified an order in the ready list as a unique order if it was not on the same route as any

one of the orders in the processing list (even though we meant to check it with all the orders in the processing list) and this sometimes resulted in having multiple orders with the same start and destination stations in the processing list. Since orders in the processing list were referred to by their start and destination station attributes by the MANAGER in order to uniquely identify them, duplicate orders were never offered to shuttles for bidding since the Manager assumes that they have already been offered.

We have not only corrected the corrupted **move_unique_orders** subroutine algorithm, but also decided to refer to orders in the processing list with a unique order id instead of using their start and destination station attributes in order to uniquely identify them. The pseudo code of the corrected algorithm **move_unique_orders** subroutine is shown in the figure below.

```
MOVE_UNIQUE_ORDERS:
for (each order B in ready_list) {
        for (each order A in processing_list) {
                if (B is NOT on the same route with A)
            continue
          else
            break;
        }
        if (END OF processing_list)
                then move B to processing_list
        if (processing_list.SIZE EQUALS TO number of shuttles alive)
                break;
}
```

Figure 24. Corrected Pseudo Code for **move_unique_orders()** Sub-routine

After debugging the subroutine and using unique order id's as unique order identifiers instead of using their start and destination station attributes, we have rerun the same test driver and observed that the visual statistical data is as expected as shown in the figure below.

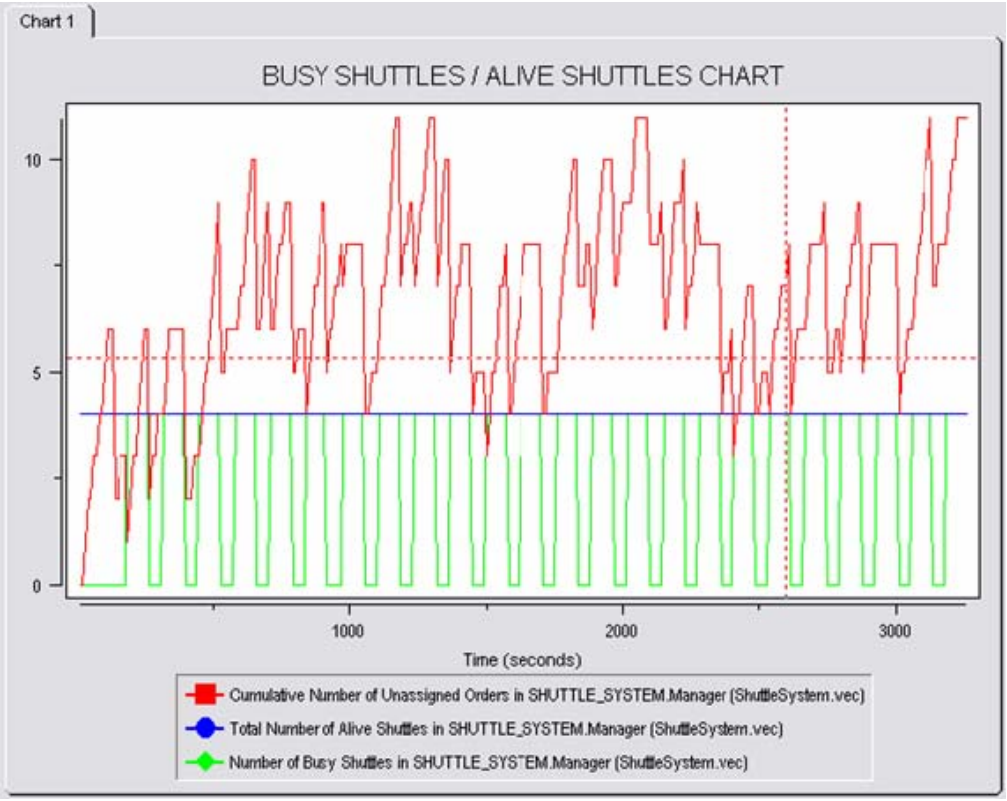

Figure 25. Visual Statistical Result of Experiment 1 (Busy Shuttles / Alive Shuttles)

2. Experiment Two

Exploring the visual statistical data gathered from our second experiment run with the AEG-based manually generated test driver with the environmental parameters displayed in Table 2, we have identified another error in the Manager module.

Test Driver No	Number of Shuttles	Number of Order Iterations by Shuttles	Customer Arrival Rate
1	4	25	Every 10 sec. with P(70)

Table 2. Parameters of AEG Based Test Driver for Experiment Two

Since the capacity of each shuttle in the system is limited to a constant number of customers (SHUTTLE CAPACITY limit for this experiment was three), the MANAGER program (control software of the shuttle system) should not be overloading shuttles by assigning customers more than the SHUTTLE CAPACITY limit of shuttles. However, as shown in the figure below, we observed

in our experiment that two of the shuttles were overloaded by the MANAGER who assigned four customers for those two shuttles.

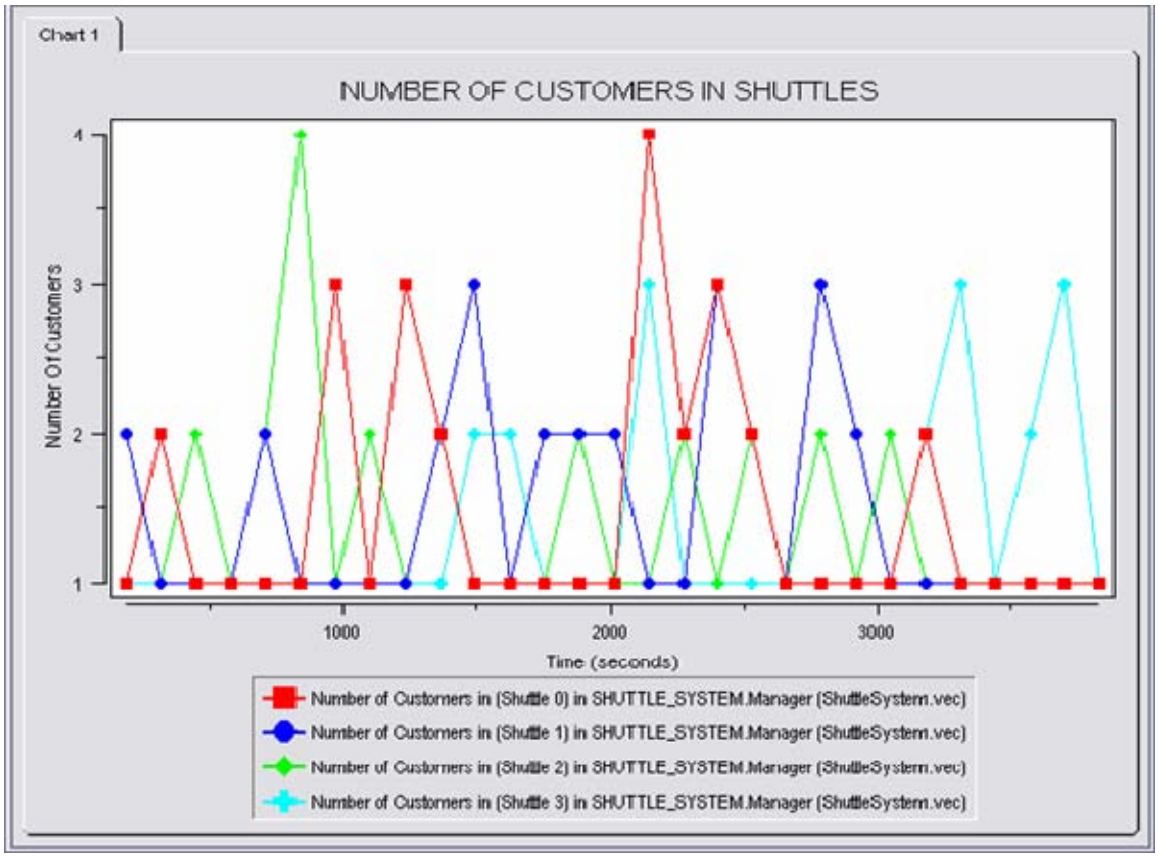

Figure 26. Visual Statistical Result of Experiment 2 (Number of Customers in Shuttles)

The cause of the problem has been found in one of the subroutines **group_matching_orders(order A)** which did not check for SHUTTLE CAPACITY limit when grouping orders. A group of orders are those orders that are on the same route. The defective algorithm (Figure 29) and the corrected algorithm (Figure 30) are shown below.

```
GROUP_ORDERS(Order A):
    for (each order B in ready_list)
    {
        if (B is on the same route with A)
        then group B with A
    }
```

Figure 27. Pseudo Code for Defective **group_matching_orders(order A)** Sub-routine

```
GROUP_ORDERS(Order A):
        if (number of orders grouped with order A is
                LESS THAN SHUTTLE CAPACITY)
        for (each order B in ready_list)
        {
            if (B is on the same route with A)
            then group B with A
            if (number of orders grouped with order A
                    EQUALS TO SHUTTLE CAPACITY)
            then break loop
        }
```

Figure 28. Pseudo Code for Corrected **group_matching_orders(order A)** Sub-routine

After debugging the subroutine we have rerun the same test driver and observed that the visual statistical data is as expected as shown in the figure below.

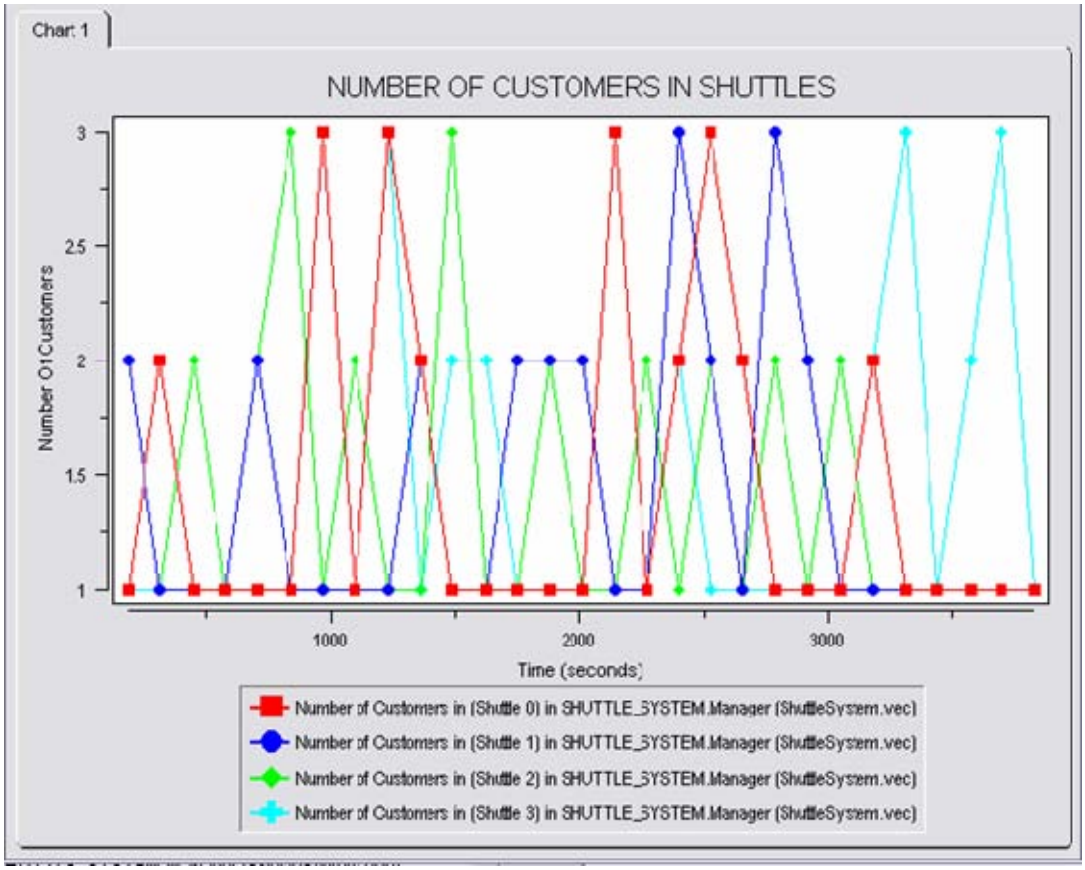

Figure 29. Visual Statistical Result of Experiment 2 (Number of Customers in Shuttles)

3. Experiment Three

We have run the manually generated test drivers shown in Table 3 and encountered another problem when we ran the last test driver where we increased the number of shuttles in the system compared to the previous test drivers. In this case our experiment simulating the interaction between the MANAGER and its environment terminated unexpectedly.

Test Driver No	Number of Shuttles	Number of Iterations of Order Processing by Shuttles	Customer Arrival Rate
1	4	15	Every 10 sec. with P(70)
2	4	25	Every 10 sec. with P(70)
3	4	30	Every 10 sec. with P(70)
4	5	20	Every 10 sec. with P(70)

Table 3. Parameters of AEG Based Test Drivers for Experiment Three

We have found that this type of error appeared because the MANAGER grouped a primary order (an order which has the longest distance in a group of orders) under another order after offering that order to the shuttles for bidding. Because of this, the MANAGER could no longer refer to that order as a primary order when shuttles sent their bids back to the MANAGER.

We have added another flag (**order.locked** attribute) to orders to prevent grouping an order that has already been offered to shuttles with another order.

After identifying and correcting each error in software correctness testing we have rerun all the previously generated test drivers. We did so because we have made changes to the SUT and it needs to be tested from scratch to find out whether we have introduced new errors to the SUT. This issue addresses regression testing [6] which is “any repetition of tests (usually after software or data change) intended to show that the software’s behavior is unchanged except insofar as required by the change to the software or data [1].” Since changes made to the SUT did not require us to change the AEG model, we have saved and reused the previously generated test drivers. This proved to be very useful for regression testing [6].

We have run the manually generated test drivers shown below in Table 4 after the last correction to the SUT and we did not identify any more errors in the SUT.

Test drivers 10 through 32 were generated by changing the values of additional customer fee and shuttle capacity variables of test driver 9 at run-time.

Test Driver No	Number of Shuttles	Number of Order Processing Iterations by Shuttles	Customer Arrival Rate	Additional Customer Fee	Shuttle Capacity
1	4	15	Every 10 sec. with P(70)	2	3
2	4	25	Every 10 sec. with P(70)	2	3
3	4	30	Every 10 sec. with P(70)	2	3
4	2	20	Every 10 sec. with P(70)	2	3
5	3	20	Every 10 sec. with P(70)	2	3
6	4	20	Every 10 sec. with P(70)	2	3
7	5	20	Every 10 sec. with P(70)	2	3
8	6	20	Every 10 sec. with P(70)	2	3
9	4	25	Every 5 sec. with P(90)	2	1
10	4	25	Every 5 sec. with P(90)	2	2
11	4	25	Every 5 sec. with P(90)	2	3
12	4	25	Every 5 sec. with P(90)	2	4
13	4	25	Every 5 sec. with P(90)	2	5
14	4	25	Every 5 sec. with P(90)	2	6
15	4	25	Every 5 sec. with P(90)	2	7
16	4	25	Every 5 sec. with P(90)	2	8
17	4	25	Every 5 sec. with P(90)	2	9
18	4	25	Every 5 sec. with P(90)	2	10
19	4	25	Every 5 sec. with P(90)	2	11
20	4	25	Every 5 sec. with P(90)	2	12
21	4	25	Every 5 sec. with P(90)	2	13
22	4	25	Every 5 sec. with P(90)	2	14
23	4	25	Every 5 sec. with P(90)	2	15
24	4	25	Every 5 sec. with P(90)	4	3
25	4	25	Every 5 sec. with P(90)	6	3
26	4	25	Every 5 sec. with P(90)	8	3
27	4	25	Every 5 sec. with P(90)	10	3
28	4	25	Every 5 sec. with P(90)	12	3
29	4	25	Every 5 sec. with P(90)	14	3
30	4	25	Every 5 sec. with P(90)	16	3
31	4	25	Every 5 sec. with P(90)	18	3
32	4	25	Every 5 sec. with P(90)	20	3
33	4	20	Every 5 sec. with P(10)	2	3
34	4	20	Every 5 sec. with P(20)	2	3
35	4	20	Every 5 sec. with P(30)	2	3
36	4	20	Every 5 sec. with P(40)	2	3
37	4	20	Every 5 sec. with P(50)	2	3
38	4	20	Every 5 sec. with P(60)	2	3
39	4	20	Every 5 sec. with P(70)	2	3
40	4	20	Every 5 sec. with P(80)	2	3
41	4	20	Every 5 sec. with P(90)	2	3

Table 4. Parameters of AEG Based Test Drivers for Experiment Three

B. SYSTEM PERFORMANCE ASSESSMENT

The purpose of system performance assessment is to get some useful information about the effectiveness of the SUT with respect to environmental variables in order to identify the hazardous situations that the system may encounter.

For the Paderborn Shuttle System, we have identified customer waiting times and shuttles' capital statuses as two mission critical variables with which we might experiment on the effectiveness of the SUT to see the correlation between these variables and other environmental parameters in order to find out if they might cause hazardous situations in the system.

1. Experiment Four

In our first experiment for system performance assessment we wanted to see the correlation between the additional customer fees and shuttles' capital statuses.

We think that shuttle capitals are mission critical values for the Paderborn Shuttle System since the low shuttle capital values indicate imminent shuttle bankruptcies which will result in decreased number of active shuttles in the system and larger customer queues. (Larger customer queues might imply longer customer waiting times and experiments concerning customer waiting times will be analyzed in Experiment 5.)

The additional customer fee is the money shuttles receive in addition to their initial bid amounts, for each additional customer (except for the one customer who is traveling the longest distance and for whom the shuttles make their bids) assigned to shuttles by the MANAGER.

We have run the test drivers shown in Table 5 below for our experiment.

Test Driver No	Number of Shuttles	Number of Order Processing Iterations by Shuttles	Customer Arrival Rate	Additional Customer Fee	Shuttle Capacity
1	4	25	Every 5 sec. with P(90)	2	3
2	4	25	Every 5 sec. with P(90)	4	3
3	4	25	Every 5 sec. with P(90)	6	3
4	4	25	Every 5 sec. with P(90)	8	3
5	4	25	Every 5 sec. with P(90)	10	3
6	4	25	Every 5 sec. with P(90)	12	3
7	4	25	Every 5 sec. with P(90)	14	3
8	4	25	Every 5 sec. with P(90)	16	3
9	4	25	Every 5 sec. with P(90)	18	3
10	4	25	Every 5 sec. with P(90)	20	3

Table 5. Parameters of AEG Based Test Drivers for Experiment Four

Initially each shuttle in the system has 100 units of money. We have considered any amount below 60 for minimum shuttle capital and any amount below 100 for average shuttle capital values as a concern of risk at the end of the time duration in which shuttles have processed at most 25 orders. The correlation between a range of additional customer fees (2 to 20 incremented by 2 each time) and minimum and average shuttle capitals are shown in the figures below.

Since we have identified any minimum shuttle capital value below 60 as a mission critical concern for the Paderborn Shuttle System, after analyzing the figure below, we may reach the conclusion that additional customer fee values of 2 and 4 may cause some shuttles to have capital balances below 60 which will be a mission risk for the system. In other terms, the minimum additional customer fee for all shuttles not to have a balance below 60 is 6 and any value above 6 will increase the performance of the overall system in terms of shuttle capital values.

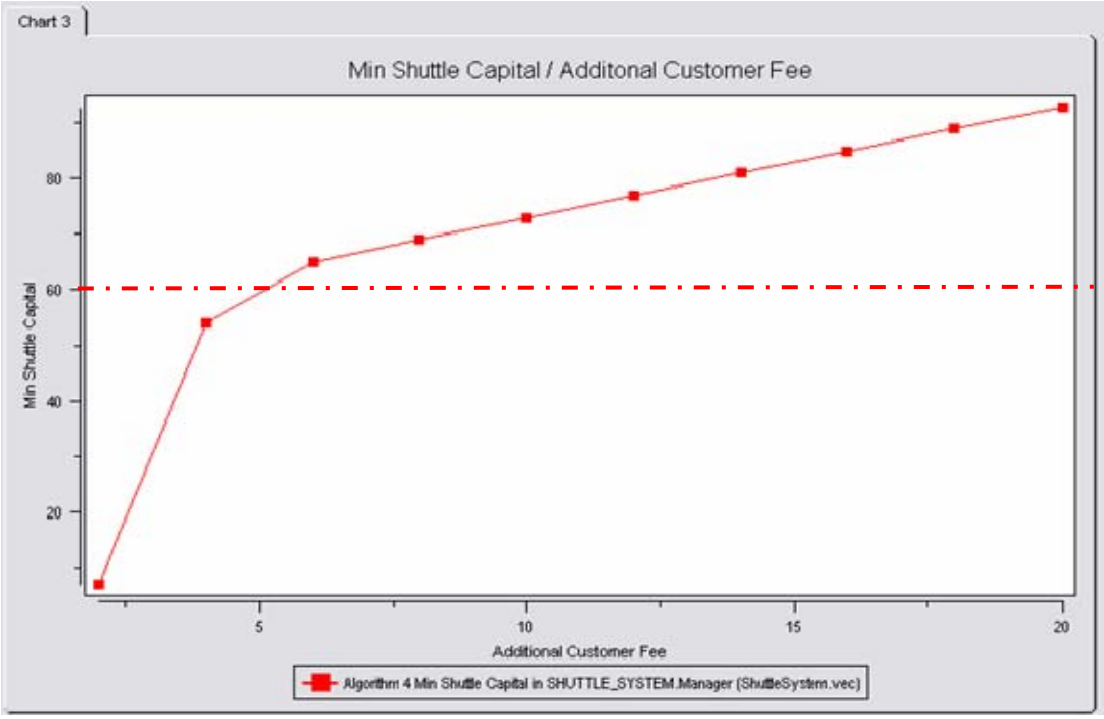

Figure 30. Visual Statistical Result of Experiment 5 (Minimum Shuttle Capital / Additional Customer Fee)

From the figure below, since we have identified any average shuttle capital values below 100 as a mission critical concern, we may reach the conclusion that additional customer fee values of 2 and 4 may cause some shuttles to have capital balances below 100 on average resulting in a system performance concern. We can also conclude that the minimum additional customer fee for all shuttles not to have an average balance below 100 is 6 and any value above 6 will increase the performance of the overall system in terms of shuttle capital values.

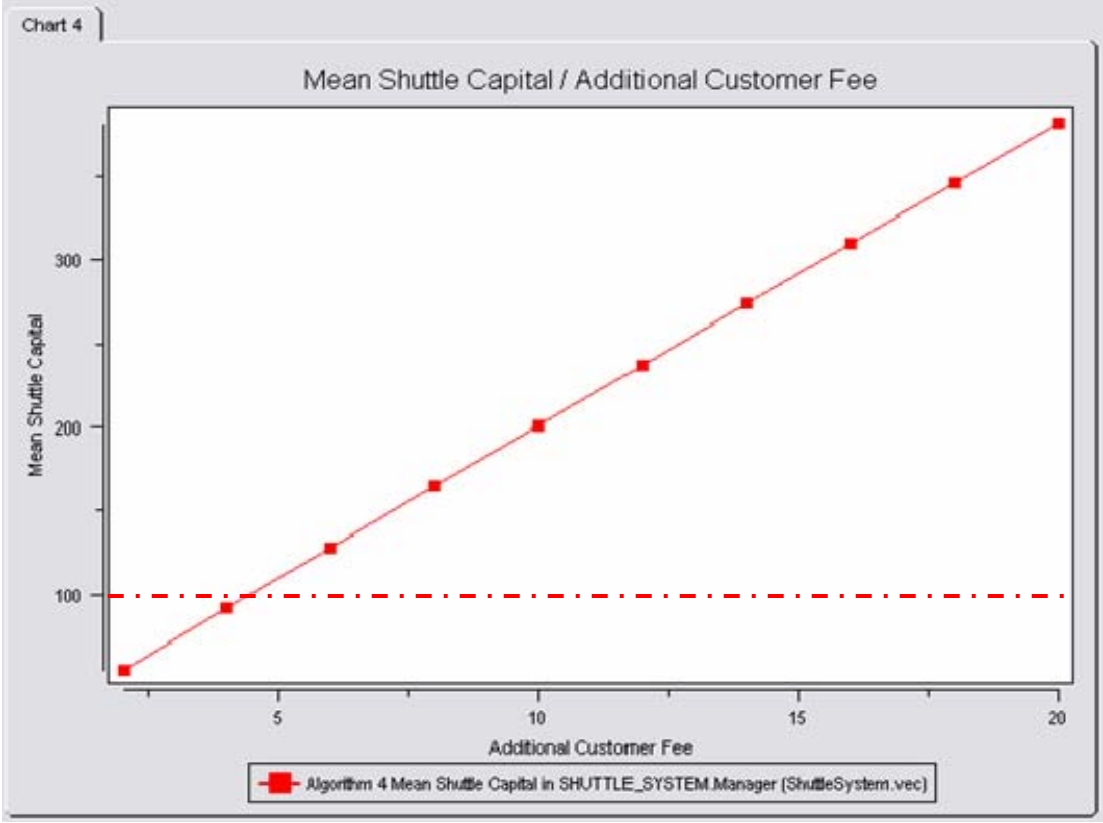

Figure 31. Visual Statistical Results of Experiment 5 (Average Shuttle Capital / Additional Customer Fee)

2. Experiment Five

In our second performance assessment experiment, we wanted to see the correlation between shuttle capacities in the system and customer waiting times in order to find out if the shuttle capacities have any effect on customer waiting times and if so, what values of shuttles capacities may cause unsafe operating conditions. We have thought that customer waiting times would be of great concern in terms of system performance since customers would not wait indefinitely and would eventually leave.

Shuttle capacity determines the maximum number of customers that can be loaded to any given shuttle in the system. We have run the test drivers shown in Table 6 below for our experiment.

Test Driver No	Number of Shuttles	Number of Order Processing Iterations by Shuttles	Customer Arrival Rate	Additional Customer Fee	Shuttle Capacity
1	4	25	Every 5 sec. with P(90)	2	1
2	4	25	Every 5 sec. with P(90)	2	2
3	4	25	Every 5 sec. with P(90)	2	3
4	4	25	Every 5 sec. with P(90)	2	4
5	4	25	Every 5 sec. with P(90)	2	5
6	4	25	Every 5 sec. with P(90)	2	6
7	4	25	Every 5 sec. with P(90)	2	7
8	4	25	Every 5 sec. with P(90)	2	8
9	4	25	Every 5 sec. with P(90)	2	9
10	4	25	Every 5 sec. with P(90)	2	10
11	4	25	Every 5 sec. with P(90)	2	11
12	4	25	Every 5 sec. with P(90)	2	12
13	4	25	Every 5 sec. with P(90)	2	13
14	4	25	Every 5 sec. with P(90)	2	14
15	4	25	Every 5 sec. with P(90)	2	15

Table 6. Parameters of AEG Based Test Drivers for Experiment Five

Our assumption is that any customer waiting more than 1500 seconds and customer waiting times over 700 seconds on average would be a concern for the time duration in which shuttles have processed at most 25 orders. The correlation between a range of shuttle capacities (1 to 15 incremented by 1 each time) and maximum and average customer waiting times are shown below in the figures below.

Since we have assumed that any maximum customer waiting time value above 1500 seconds is a mission critical value, we may reach the conclusion, from the figure below, that shuttle capacity values of 1, 2 and 3 may cause some customers to wait more than 1500 seconds. Stated differently, the minimum shuttle capacity value for all shuttles in order to prevent any customer from waiting more than this critical level of 1500 seconds is 4 and any value above 4 will increase the performance of the overall system decreasing the customer waiting times.

Figure 32. Visual Statistical Result of Experiment 5 (Maximum Customer Waiting Time (seconds) / Shuttle Capacity)

Having identified any average customer waiting time value above 700 seconds as a mission critical value, from the figure below, we may conclude that shuttle capacity values of 1, 2 and 3 may cause some customers to wait more than 700 seconds on average. We can also state that the minimum shuttle capacity value for all shuttles not to cause customers to wait more than this critical level of 700 seconds average waiting time is 4 and any value above 4 will increase the performance of the overall system in terms of customer waiting times.

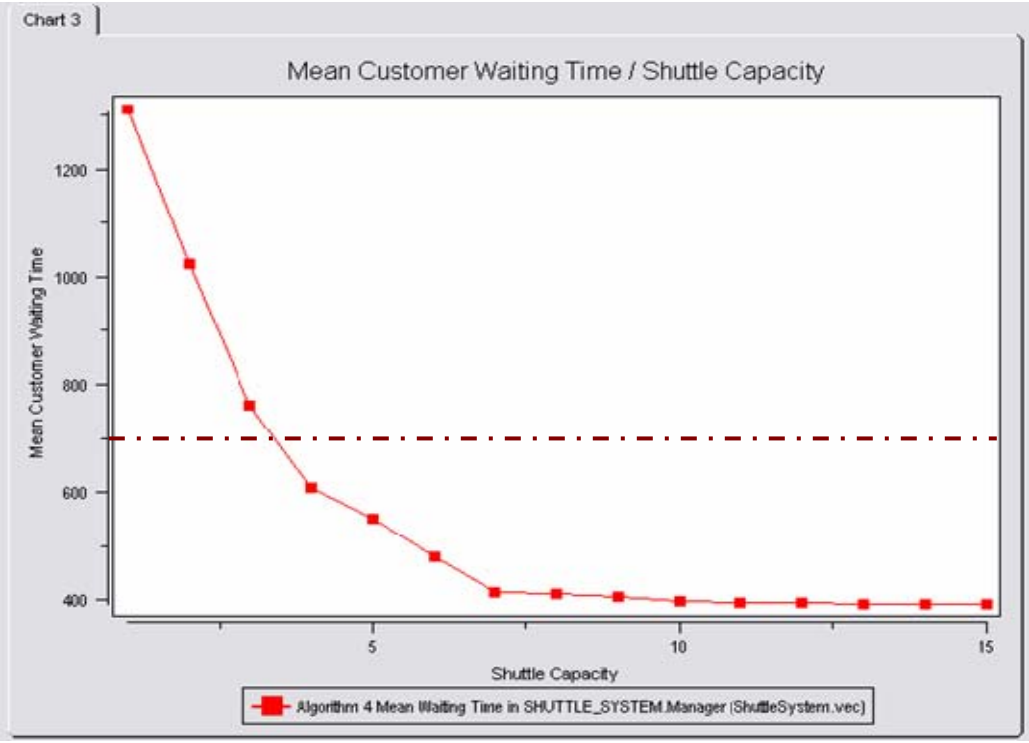

Figure 33. Visual Statistical Result of Experiment 5 (Average Customer Waiting Times (seconds) / Shuttle Capacity)

C. EVALUATION OF DESIGN ALTERNATIVES

This experiment arose because we wanted to explore the efficiency of four different algorithms that are used for choosing primary orders in the context of its operating environment. Without this type of experiment we cannot reach to a conclusion regarding the efficiency of an algorithm with respect to environmental parameters. We measure the efficiency of those four different algorithms for this purpose by running them with the same scenario batches for each algorithm and comparing the statistics gathered from those runs.

What we are really interested in this experiment is observing and comparing the efficiency of each algorithm under different environmental factors.

The general algorithm for choosing orders, grouping orders and assigning them to shuttles is as follows: Choose primary orders from a queue of order requests waiting to be processed and then group the rest of the orders with primary orders if they are on the same route as primary orders as long as the

number of orders grouped with primary orders does not exceed shuttle capacity limits.

We have developed four different algorithms to choose primary orders where the chosen primary orders are moved from the ready list to the processing list.

The term unique orders used in the algorithms described in the figures below are those orders where the path (a sequence of stations on a shuttle's route in order to complete its assignment) of an order is neither the same as the path of a primary order which is in the processing list nor is it a subset path of a primary order.

```
CHOOSE PRIMARY ORDERS (algorithm 1):
        MOVE_UNIQUE_ORDERS
```

Figure 34. Algorithm 1 for Choosing Primary Orders

```
CHOOSE PRIMARY ORDERS (algorithm 2):
        while((processing_list SIZE < number of shuttles alive) AND
          (ready_list NOT EMPTY)) {
      MOVE_UNIQUE_ORDERS
      if(processing_list SIZE NOT CHANGED)
          break;
      for(each order in the processing_list) {
          if(an order is on the same route with another order in the list)
              group that order with its superset order
      }
      if(processing_list SIZE NOT CHANGED)
          break;
  }
```

Figure 35. Algorithm 2 for Choosing Primary Orders

```
CHOOSE PRIMARY ORDERS (algorithm 3):
        while((processing_list SIZE < number of shuttles alive) AND
          (ready_list NOT EMPTY))
      move the first order from the processing_list to ready_list
```

Figure 36. Algorithm 3 for Choosing Primary Orders

```
CHOOSE PRIMARY ORDERS (algorithm 4):
        while((processing_list SIZE < number of shuttles alive) AND
          (ready_list NOT EMPTY)) {
      move the first order from the processing_list to ready_list
      if(processing_list SIZE NOT CHANGED)
          break;
      for(each order in the processing_list) {
          if(an order is on the same route with another order in the list)
              group that order with its superset order
      }
      if(processing_list SIZE NOT CHANGED)
          break;
  }
```

Figure 37. Algorithm 4 for Choosing Primary Orders

We have compared each algorithm changing the values of three environment parameters, one at a time: number of shuttles in the system, shuttle capacities and customer arrival rates in order to measure their efficiencies in terms of efficiency parameters such as average shuttle capacity, average shuttle capital and minimum, maximum and average customer waiting times.

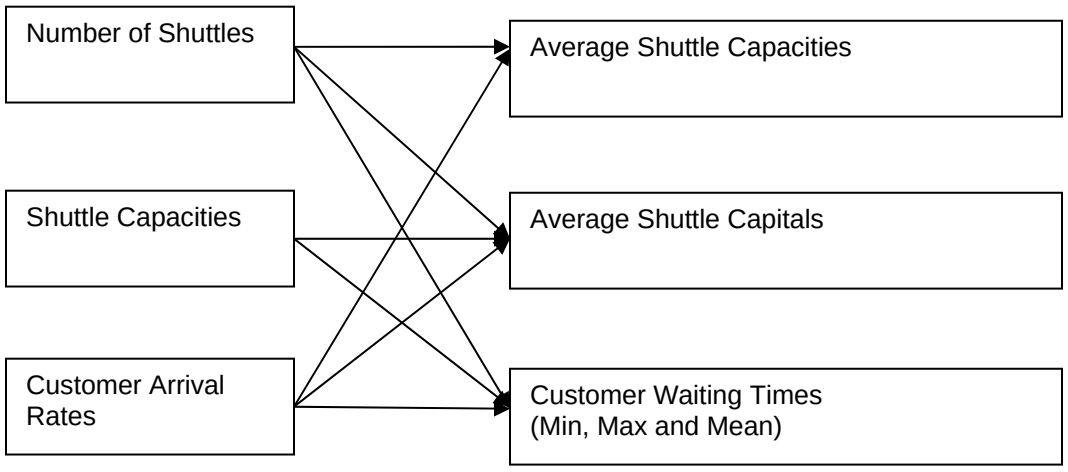

Figure 38. System Parameters of Interest In Order To Measure the Efficiency of Different Algorithms

1. Experiment Six

In our first experiment of evaluation of design alternatives, we have explored the efficiency of four different algorithms (each displayed with a different color in the figures below) changing the number of shuttles in the system. We have measured the efficiency of different algorithms in terms of average shuttle capacity utilization, average shuttle capital and minimum, maximum and average customer waiting times.

This experiment was conducted using the AEG based test drivers displayed in the table below.

Test Driver No	Number of Shuttles	Number of Order Processing Iterations by Shuttles	Customer Arrival Rate	Additional Customer Fee	Shuttle Capacity
1	2	25	Every 5 sec. with P(90)	2	3
2	3	25	Every 5 sec. with P(90)	2	3
3	4	25	Every 5 sec. with P(90)	2	3
4	5	25	Every 5 sec. with P(90)	2	3

Table 7. Parameters of AEG Based Test Drivers for Experiment Six

From the first figure below, we can easily make a general conclusion that increasing the number of shuttles in the system results in decreasing shuttle capacity utilization. We can also find out which algorithm is more efficient in terms of shuttle capacity utilization values in a couple of different cases where we have a varying number of shuttles in the system. For instance, it can be easily observed that algorithm 3 is the least efficient when there are 4 or 5 shuttles in the system since it has the lowest average shuttle capacity utilization.

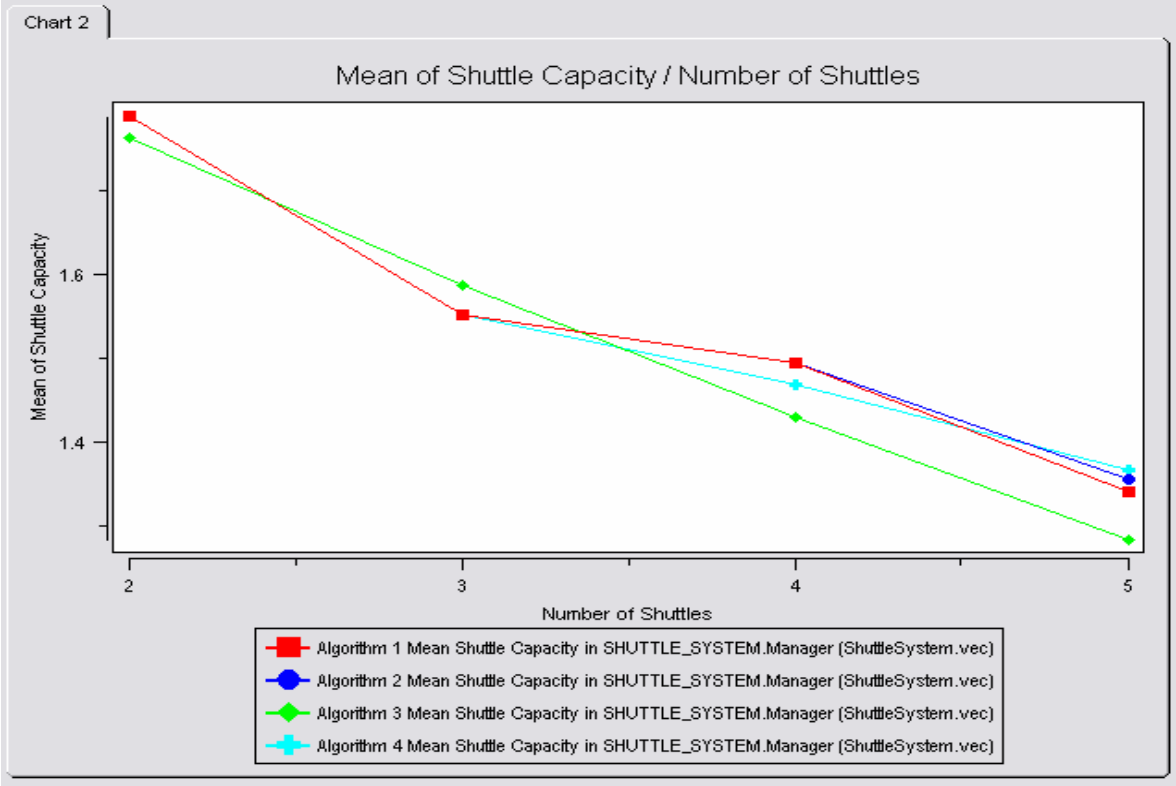

Figure 39. Visual Statistical Results of Experiment Six (Average Shuttle Capacity / Number of Shuttles)

Analyzing the next figure below, we can conclude generally that increasing the number of shuttles in the system results in decreasing average shuttle capital values. We can also find out which algorithm is more efficient in terms of average shuttle capital values in a couple of different cases where we have a varying number of shuttles in the system.

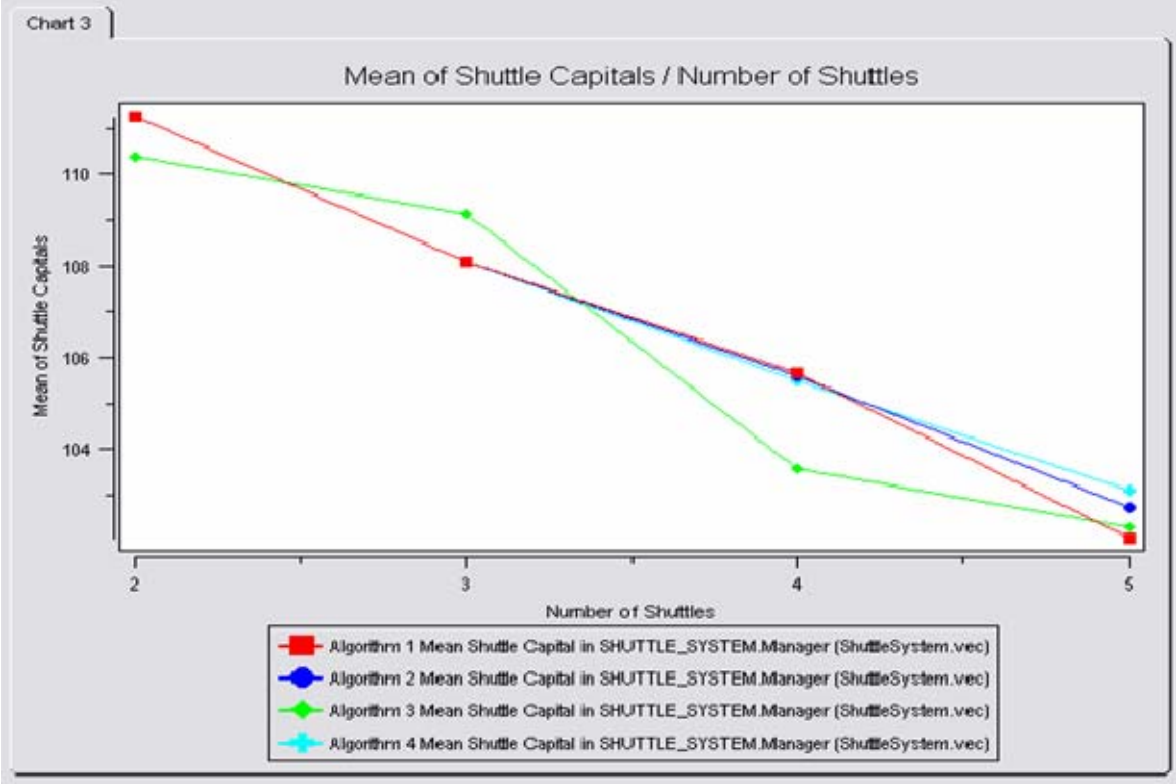

Figure 40. Visual Statistical Results of Experiment Six (Average Shuttle Capitals / Number of Shuttles)

Analyzing the three figures below, we can make a general conclusion that increasing the number of shuttles in the system helps to reduce minimum, maximum and average customer waiting times. We can also find out which algorithm is more efficient in terms of customer waiting times in a couple of different cases where we have a varying number of shuttles in the system.

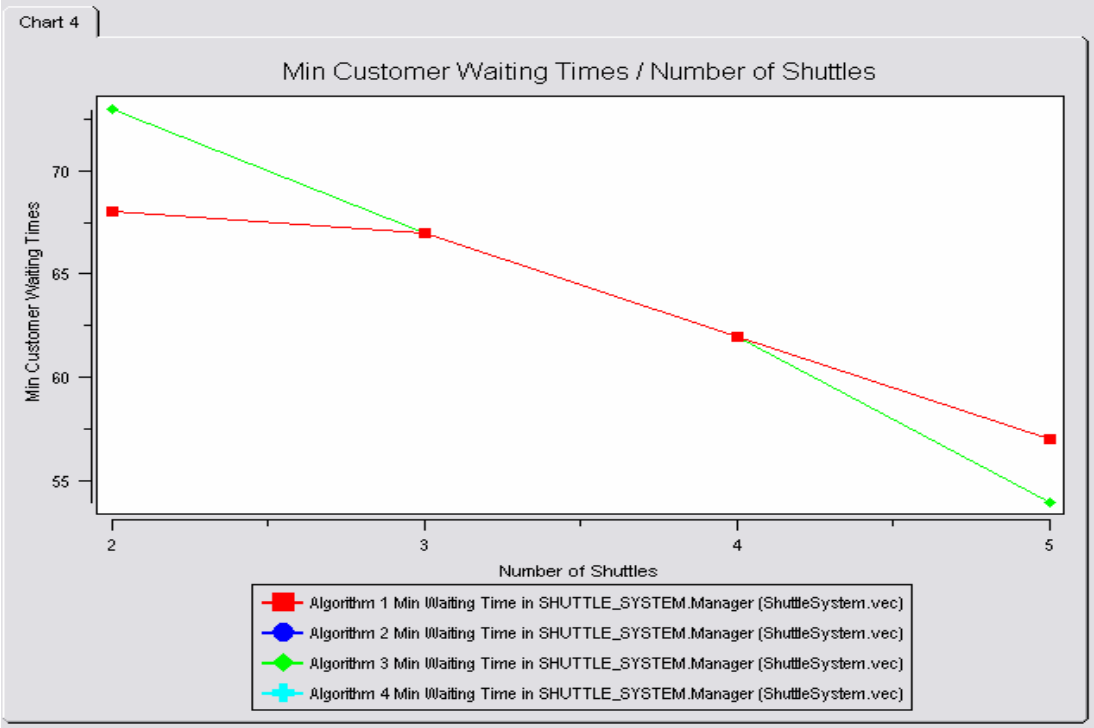

Figure 41. Visual Statistical Results of Experiment Six (Minimum Customer Waiting Times (seconds) / Number of Shuttles)

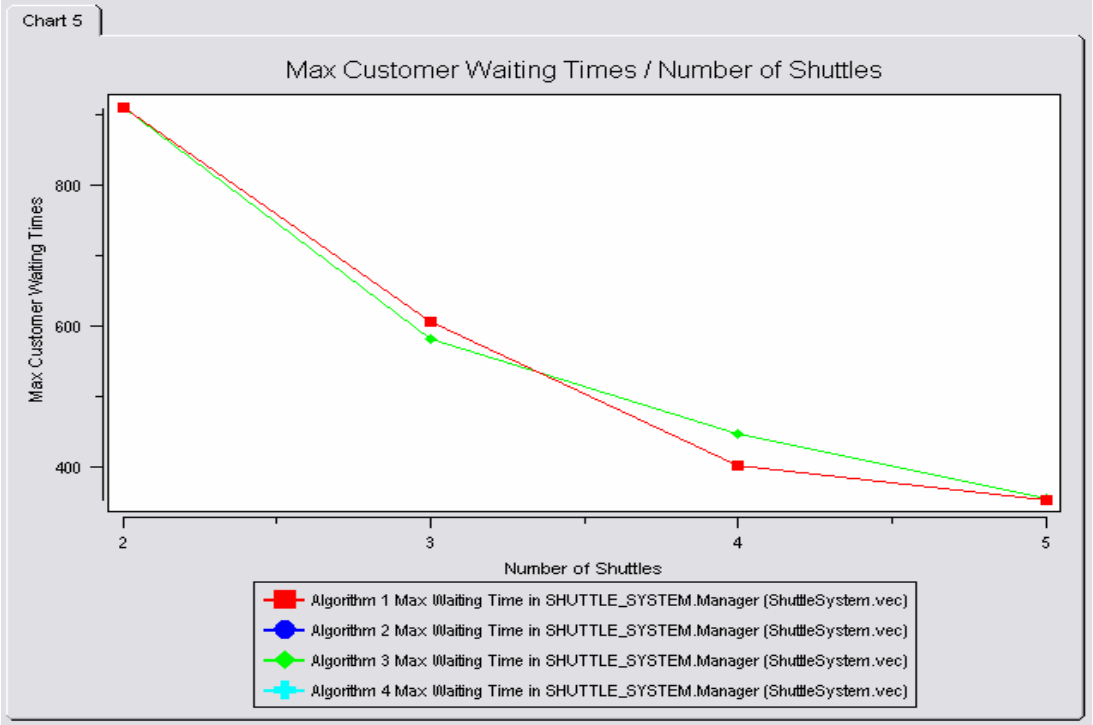

Figure 42. Visual Statistical Results of Experiment Six (Maximum Customer Waiting Times (seconds) / Number of Shuttles)

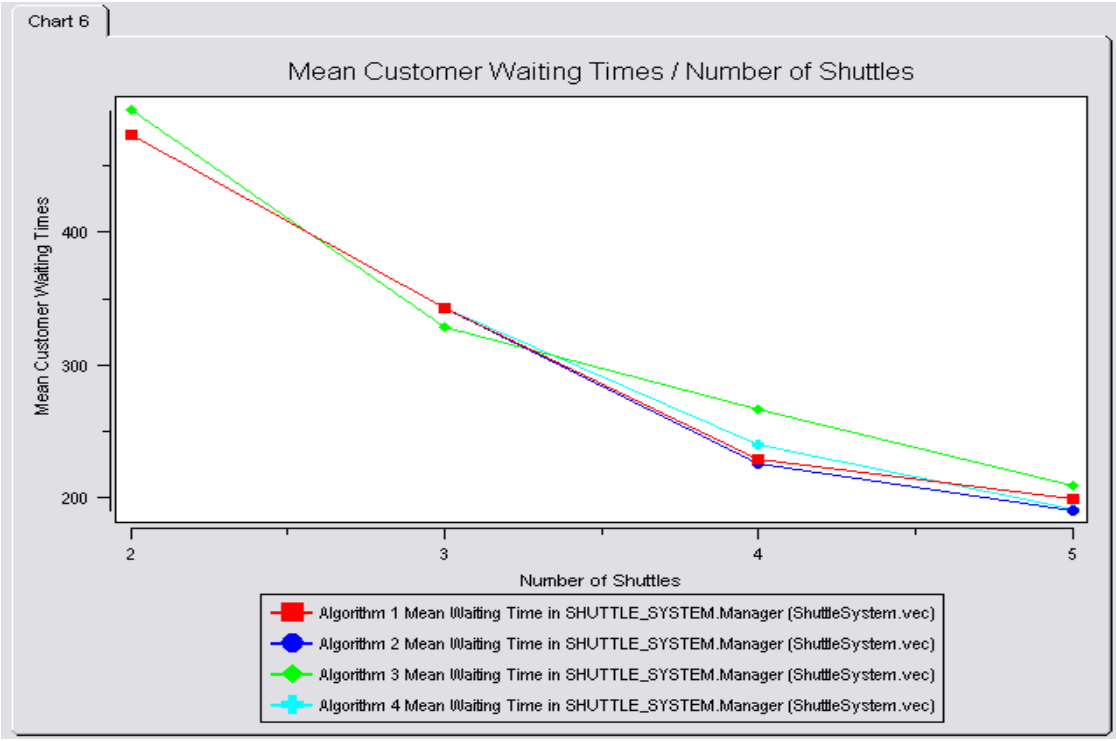

Figure 43. Visual Statistical Results of Experiment Six (Average Customer Waiting Times (seconds) / Number of Shuttles)

2. Experiment Seven

In our second experiment of evaluation of design alternatives, we have explored the efficiency of 4 different algorithms (each displayed with a different color in the figures below) changing the shuttle capacity parameter of the system. We have measured the efficiency of different algorithms in terms of average shuttle capital and minimum, maximum and average customer waiting times parameters of the system.

This experiment was conducted using the AEG based test drivers displayed in the table below.

Test Driver No	Number of Shuttles	Number of Order Processing Iterations by Shuttles	Customer Arrival Rate	Additional Customer Fee	Shuttle Capacity
1	4	25	Every 5 sec. with P(90)	2	1
2	4	25	Every 5 sec. with P(90)	2	2
3	4	25	Every 5 sec. with P(90)	2	3
4	4	25	Every 5 sec. with P(90)	2	4
5	4	25	Every 5 sec. with P(90)	2	5
6	4	25	Every 5 sec. with P(90)	2	6
7	4	25	Every 5 sec. with P(90)	2	7
8	4	25	Every 5 sec. with P(90)	2	8
9	4	25	Every 5 sec. with P(90)	2	9
10	4	25	Every 5 sec. with P(90)	2	10

Table 8. Parameters of AEG Based Test Drivers for Experiment Seven

From the figure below, we can conclude generally that increasing shuttle capacity values in the system results in increased average shuttle capital values. We can also find out which algorithm is more efficient in terms of average shuttle capital values in a couple of different cases where we have a varying number of shuttle capacity values in the system. Algorithm 3 appears to be the least efficient for each case while algorithm 2 is the most efficient one for all of the different shuttle capacities.

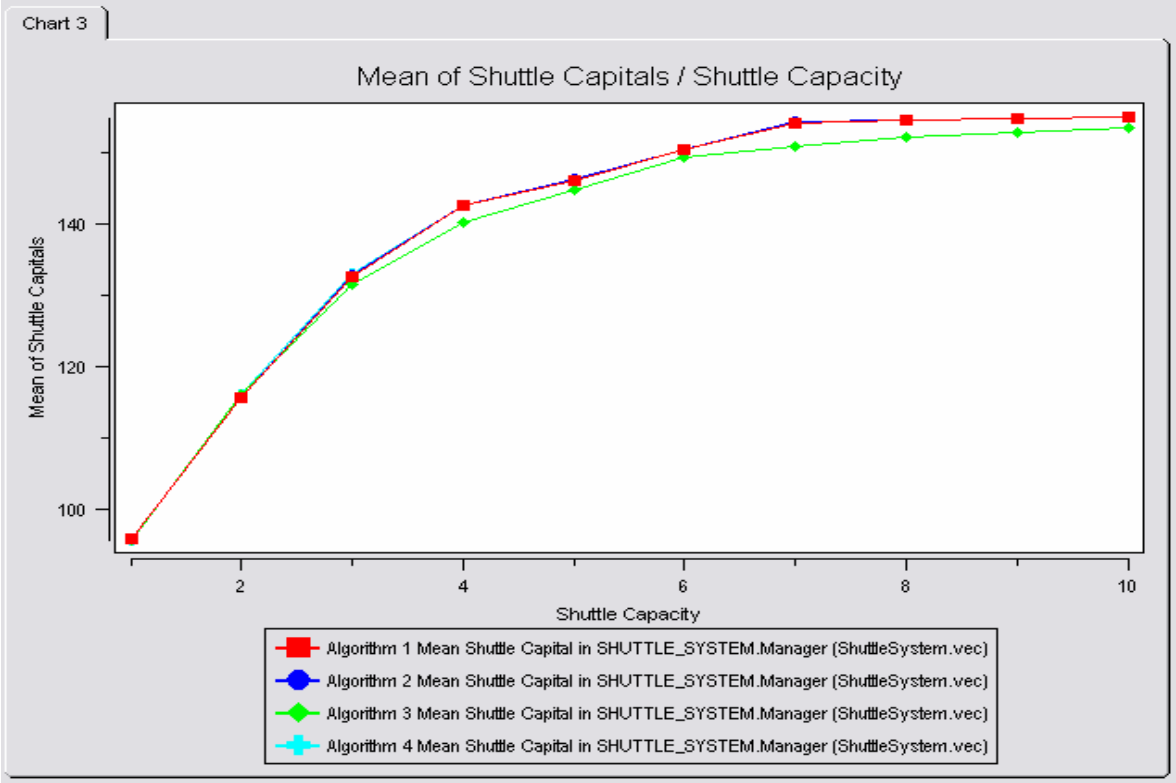

Figure 44. Visual Statistical Results of Experiment Seven (Average Shuttle Capitals / Shuttle Capacity)

From the figure below, we can conclude generally that increasing shuttle capacity values up to 4 in the system results in decreasing minimum customer waiting times and increasing the shuttle capacity above 4 has no effect on minimum customer waiting times for all algorithms. We can also find out which algorithm is more efficient in terms of minimum customer waiting times in a couple of different cases where we vary shuttle capacities in the system. Algorithm 3 appears to be the least efficient when the shuttle capacity is 3 while the rest of the algoritms yield the same better result.

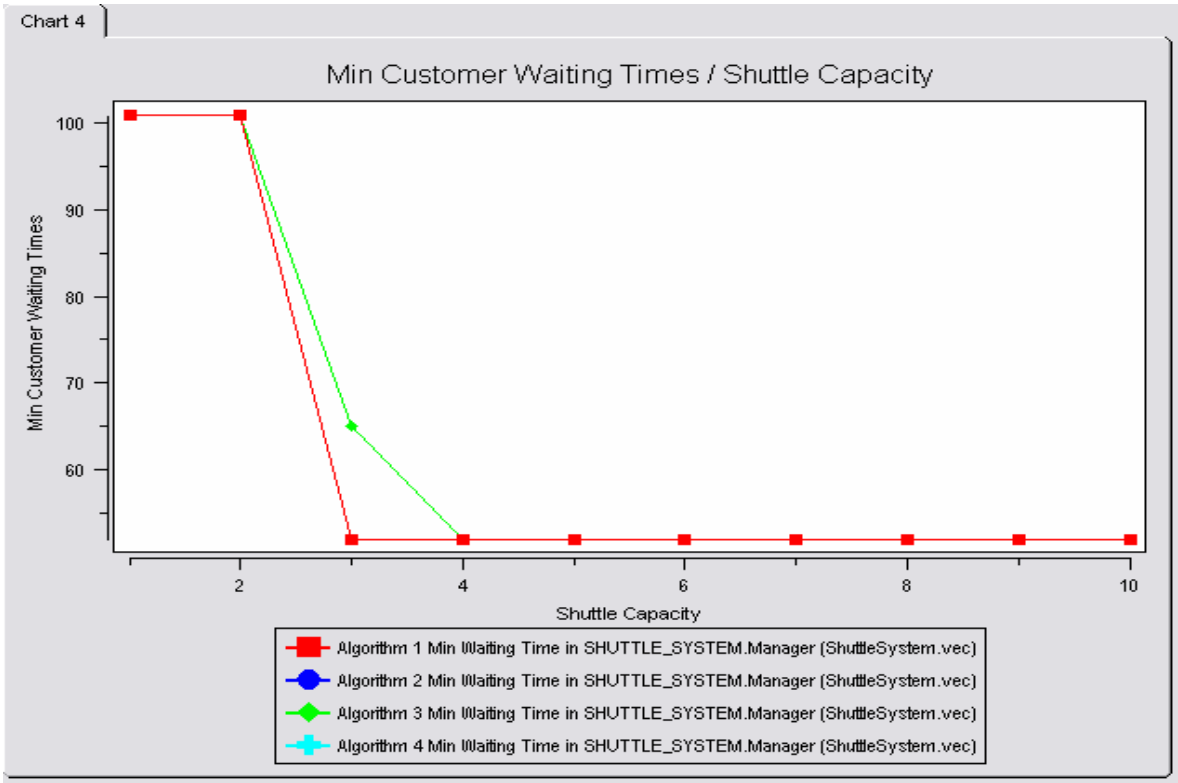

Figure 45. Visual Statistical Results of Experiment Seven (Minimum Customer Waiting Times (seconds) / Shuttle Capacity)

From the figure below, we can conclude generally that increasing shuttle capacity values up to 7 in the system results in decreasing maximum customer waiting times and increasing the shuttle capacity more than 7 has no effect on maximum customer waiting times for all algorithms. We can also find out which algorithm is more efficient in terms of maximum customer waiting times in a couple of different cases where we have a varying number of shuttle capacities in the system.

Figure 46. Visual Statistical Results of Experiment Seven (Maximum Customer Waiting Times (seconds) / Shuttle Capacity)

From the figure below, we can conclude generally that increasing shuttle capacity values results in decreasing average customer waiting times. We can also find out which algorithm is more efficient in terms of average customer waiting times in a couple of different cases where we have a varying number of shuttle capacities in the system. Algorithm 3 appears to be the least efficient for all cases of different shuttle capacities while the other algorithms have almost the same values.

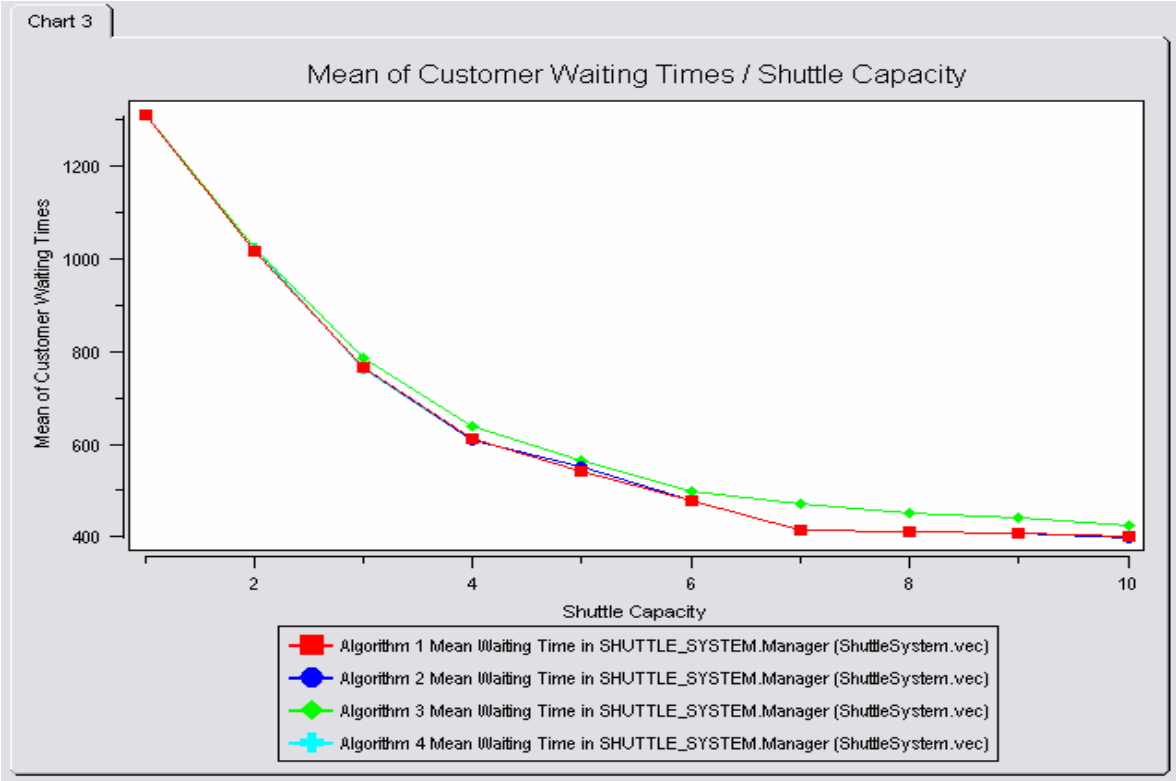

Figure 47. Visual Statistical Results of Experiment Seven (Average Customer Waiting Times (seconds) / Shuttle Capacity)

3. Experiment Eight

In our third and final experiment evaluating design alternatives, we have explored the efficiency of 4 different algorithms (each displayed with a different color in the figures below) changing the customer arrival rate parameter of the system. We have measured the efficiency of these algorithms in terms of shuttle capacity utilization, average shuttle capital and maximum and average customer waiting time parameters of the system.

This experiment was conducted using the AEG based test drivers displayed in the table below.

Test Driver No	Number of Shuttles	Number of Order Processing Iterations by Shuttles	Customer Arrival Rate	Additional Customer Fee	Shuttle Capacity
1	4	20	Every 5 sec. with P(10)	2	3
2	4	20	Every 5 sec. with P(20)	2	3
3	4	20	Every 5 sec. with P(30)	2	3
4	4	20	Every 5 sec. with P(40)	2	3
5	4	20	Every 5 sec. with P(50)	2	3
6	4	20	Every 5 sec. with P(60)	2	3
7	4	20	Every 5 sec. with P(70)	2	3
8	4	20	Every 5 sec. with P(80)	2	3
9	4	20	Every 5 sec. with P(90)	2	3

Table 9. Parameters of AEG Based Test Drivers for Experiment Eight

From the first figure below, we can easily conclude that increasing customer arrival rates results in increasing shuttle capacity utilization. We can also find out which algorithm is more efficient in terms of shuttle capacity utilization values in a couple of different cases where we have increased customer arrival rates by P(10) at a time for the range of P(10) to P(90). It can be easily observed that while algorithm 3 is the least efficient for low customer arrival rates, there is no significant difference between those algorithms as the customer arrival rate gets higher.

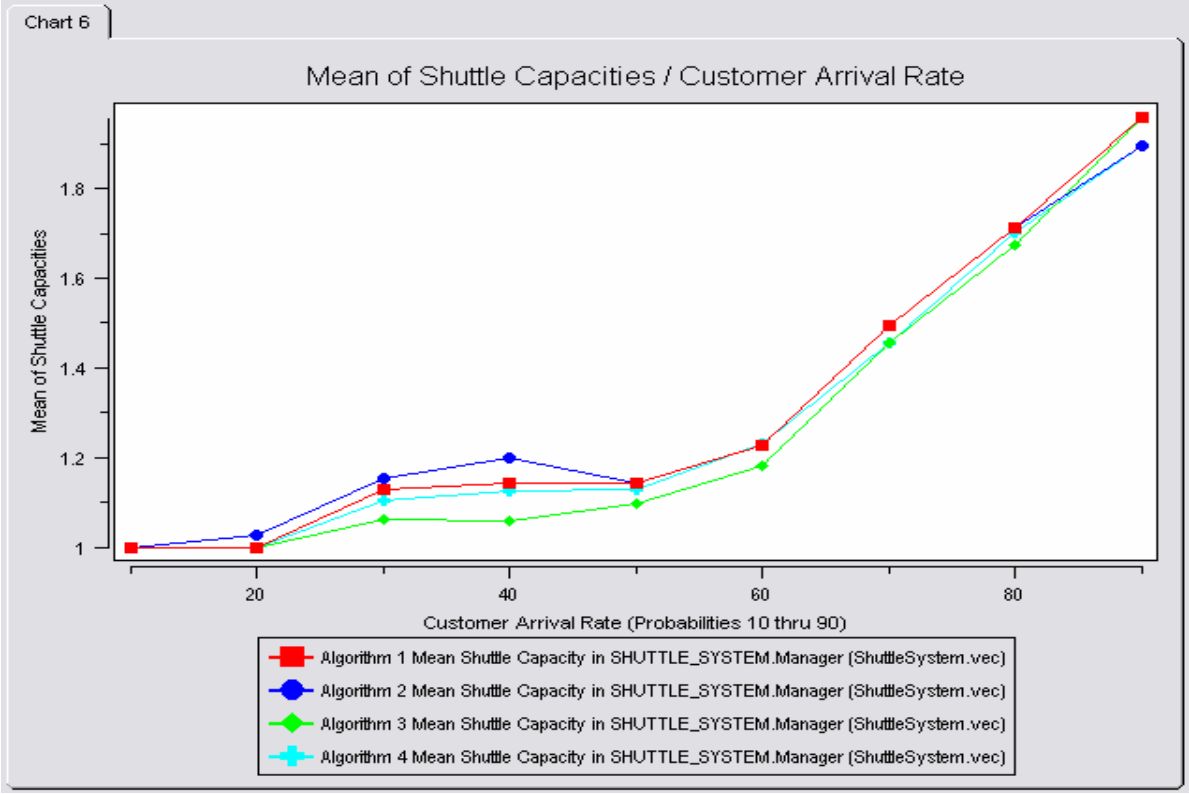

Figure 48. Visual Statistical Results of Experiment Eight (Mean Shuttle Capacities / Customer Arrival Rate)

We can conclude from the figure below that for customer arrival rate of P(10) all of the algorithms cause shuttles to have the same capital values. However, while algorithm 3 is the least efficient overall for higher customer arrival rates, algorithm 4 yields higher average shuttle capitals.

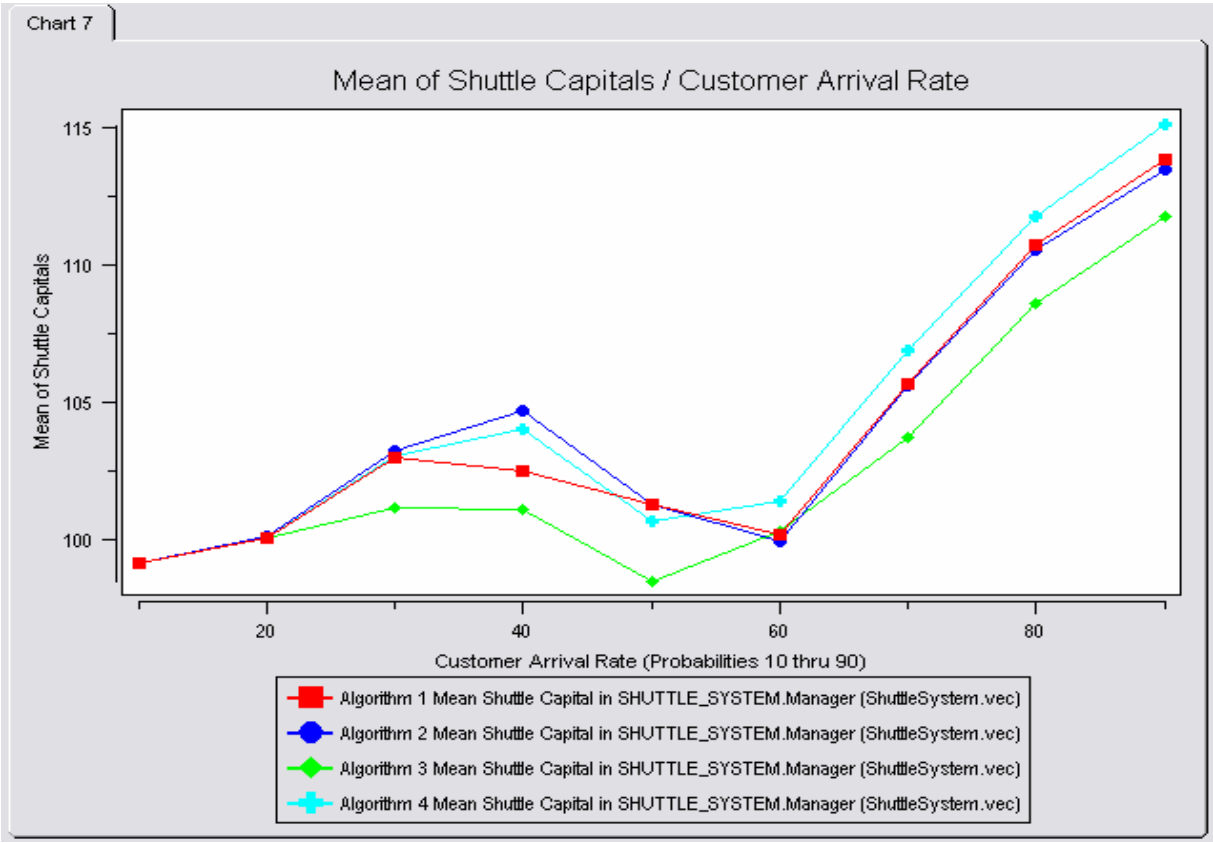

Figure 49. Visual Statistical Results of Experiment Eight (Mean Shuttle Capitals / Customer Arrival Rate)

The figure below yields rather interesting results in terms of measuring the efficiency of different algorithms with respect to maximum customer waiting times. Generally we can conclude that increasing customer arrival rates results in increasing maximum customer waiting times. In terms of comparisons of algorithm efficiencies, while all of the algorithms yields the same maximum customer waiting times up to customer arrival rate probability of P(50), after this point, algorithm 3 gives the highest maximum customer waiting times whereas algorithms 2 and 4 treats customers rather fairly, not causing any of them to wait too long.

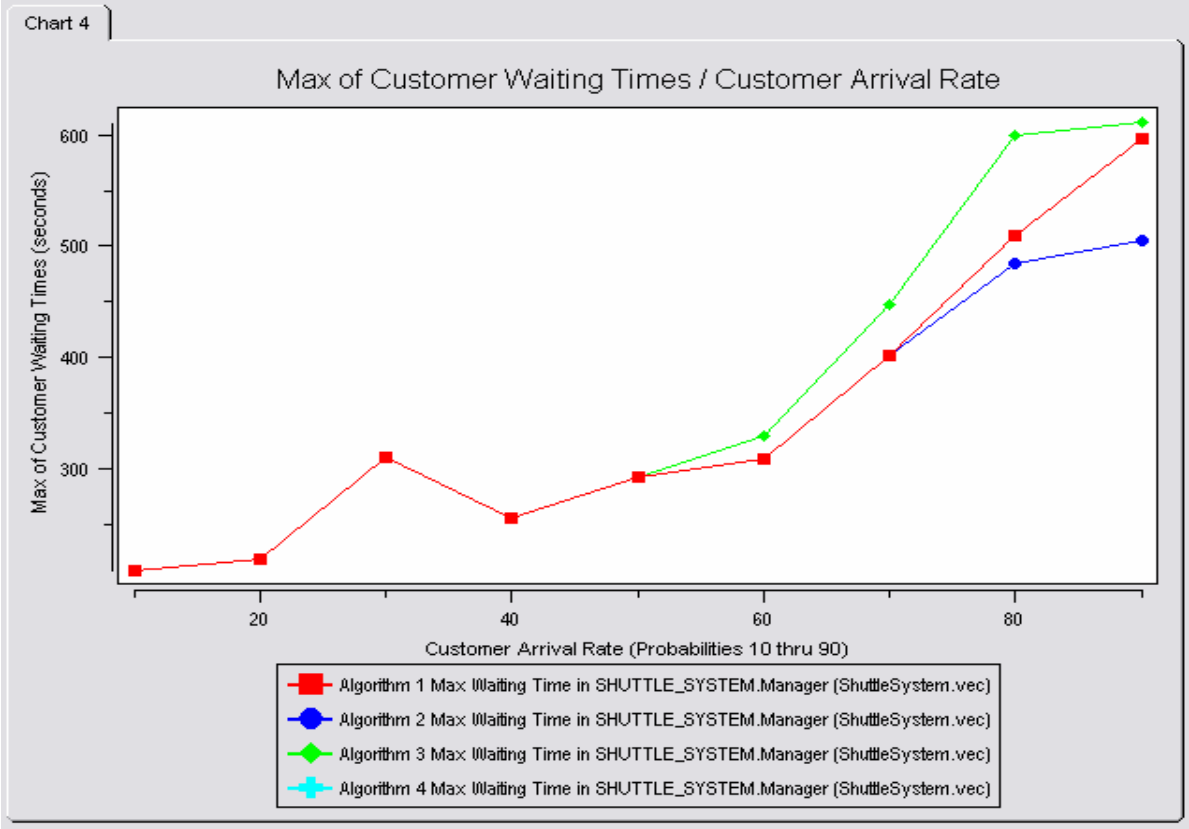

Figure 50. Visual Statistical Results of Experiment Eight (Maximum Customer Waiting Times / Customer Arrival Rate)

From the figure below, we can conclude that increasing customer arrival rates generally results in increasing average customer waiting times. In terms of algorithm efficiency comparisons, while all of the algorithms yields the same average customer waiting times at P(10), after this point, algorithm 3 gives the highest average customer waiting times whereas the other algorithms gives lower average customer waiting times with no significant difference among them.

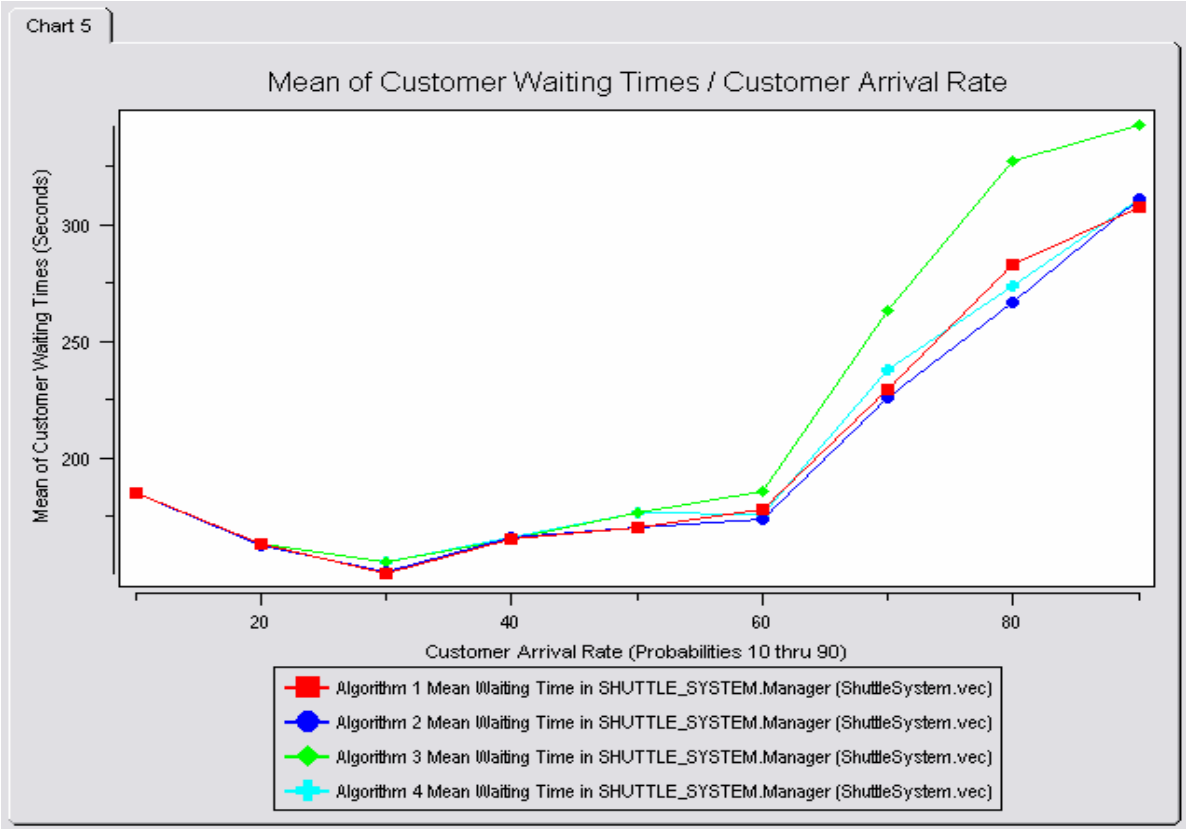

Figure 51. Visual Statistical Results of Experiment Eight (Average Customer Waiting Times / Customer Arrival Rate)

IV. RELATED WORK

Hand-crafted testing (manual testing) is the state-of-practice in today's software industry [13]. However, this approach has many shortcomings such as being tedious for the tester, expensive for the company [13], error prone, slow, and less efficient in the long run [5].

Static test automation [13] (Capture/Playback Approach [14]) is one way of automating this process. This approach still relies on manually determined test cases. Capturing the manual test sequences in a test script and rerunning them [14] for regression testing purposes at a later time is the only improvement in this approach. However, it is costly to modify test scripts when the system under testing changes [13], [14]. Moreover, exercising the same sequence of test inputs over and over again reduces the chance of finding new errors [13].

An even better improvement is using random test programs (dumb monkeys [13]) which generate test scenarios randomly and aimlessly. Even though they prove to be very useful in finding crashing bugs by generating unusual test input sequences [13], they are not systematic and cannot be directed to the specific parts of the SUT [5], [13].

A systematic and focused [5] testing automation is what we seek. Systematic testing enables us to enumerate input and state combinations [5] and measure testing coverage. Focused testing helps us to concentrate on the specific parts of the SUT where bugs might be likely to be found [5], [13].

Model-based test generation is based on a description of the application's behavior to determine what actions are possible and what outcome is expected [13] and this approach achieves the objectives explained above [5]. The SUT's behavior can be represented in a state table from which a computer can generate test sequences that are randomly selected [13] from available test sequences that are associated with the current state of the SUT.

Whittaker [15] and Maurer [17] describe a solution to determining test sequence generation problem where regular expressions and grammars can be used in addition to graph or state diagrams proposed by Robinson [13] to create a behavior model of the SUT. Using the language theory, in this approach test inputs are represented as symbols which are generated randomly according to a template that describes a test format [17]. Then they are combined to make valid words and sentences that can be applied to the SUT [15].

Blacburn, Busser and Nauman [16] state that “model-based test generation can be based on various modeling forms, such as state machines, functional tabular condition/action models, control system models, language models and hybrids.”

All of the Model-based test generation approaches mentioned above are based on modeling the behavior of the SUT whereas our approach in Environment Behavior Modeling for testing automation is totally black-box and oblivious of the behavior of the SUT. Attributed Event Grammars (AEG), which are based on context free grammars, are used to describe the events in the environment of the SUT in order to generate random test sequences that can be applied to the SUT. Moreover, our approach of automatic scenario generation from Environment Behavior Models is especially useful for testing of real-time reactive systems [6], [7].

In order to specify the externally observable behavior of a software module, Wang and Parnas [18] used trace assertions [7]. They presented a trace simulator to symbolically interpret the trace assertions and simulate the externally observable behavior of the module specified using algebraic specifications and term and term rewriting techniques [7]. This approach is only applicable to non-real-time applications [7].

Alfonso et al. [19] used event scenarios (events and responses) for expressing real-time system constraints with a formal visual language [7]. They proposed to study properties of the formal model of the system under analysis via model checking and run-time verification with a tool that could translate the

scenarios into observer timed automata [7]. However, while this approach is effective for modeling static environments with fixed scenarios, AEG based environment behavior modeling can be used to specify dynamic environments with concurrent events [7].

Randomized test cases against “usage probability distributions [20]” which aims at minimizing the test input sequence combinations for increasing the efficiency of test cases are described by Linger in the Cleanroom Software Engineering approach. Usage probability distributions which “define all possible usage patterns and scenarios, including erroneous and unexpected usage, together with their probabilities of occurrence [20]” are natural in the semantics of Event Grammars. P(prob) constructs of AEG’s can be used efficiently for such purposes. Moreover, changing environmental parameters of the test cases (either at compile time or run-time) can help direct the focus of test drivers.

Environment Behavior Models for testing automation and safety assessment of real-time reactive systems is first described by Auguston, Michael and Shing in [6] and [7] where the fundamentals of Event Grammars are explained and possible environment models for three different case studies (Calculator Program, CARA Infusion Pump System and The Paderborn Shuttle System Control Software) are shown. In [8], efficiency of Environment Behavior Models is explored in detail by the CARA Infusion Pump case study (which is a safety critical real-time reactive system) through three kinds of experiments: Quantitative Safety Assessment, Qualitative Safety Assessment and Development/Improvement of the SUT.

THIS PAGE INTENTIONALLY LEFT BLANK

V. CONCLUSION

We have explored the effectiveness of using environment behavior models as a method for testing and analyzing real-time, reactive software systems. We have used automatic test case scenario generation, which is based on an attributed event grammar (AEG) model, to define the environment of a SUT which is a real-time, reactive software system. This system was a model of the Paderborn Shuttle System [10] control software which is a real-time reactive system. We have explored the extent to which experiments with a SUT embedded in an environment behavior model serve as a constructive method for testing functional, performance and timing requirements of real-time, reactive software systems.

Specifically, we have conducted three types of experiments to investigate the effectiveness of the AEG-based test automation: software correctness testing, system performance assessment and evaluation of design alternatives for the control software.

The experience gained from our case study of the Paderborn Shuttle System [10] control software together with the experiences learned from the previously exercised case study of CARA [8] control software reveals the main advantages of the approach as follows:

- “AEG is well structured and hierarchical, as is any formalism based on formal grammars [7].”
- “Environment models specified by attributed event grammars provide for automated generation of a large number of random test drivers [7].”
- Different environment models can be developed throughout the design process (especially when the spiral method is used) for unit and system testing purposes. We have developed our SUT (Paderborn Shuttle System control software) with the spiral

software development method and used appropriate environment models for the prototypes generated at the end of each development cycle in order to test the SUT at each level. Environment models proved very useful in identifying both serious requirements errors and minor coding errors in the SUT throughout the software development process.

- Generated test drivers can be used for real-time system correctness testing purposes as demonstrated in experiments one thru three. Our experiments proved very useful in identifying critical errors in the SUT which would be very hard to find without the use of the AEG based automated scenario generation approach.
- Extreme case test scenarios can be generated from attributed event grammar based environment models for load testing of the system.
- Usage probability distributions which “define all possible usage patterns and scenarios, including erroneous and unexpected usage, together with their probabilities of occurrence [20]” are natural in the semantics of Event Grammars. P(prob) constructs of AEG’s can be used efficiently for such purposes.
- “Generated test drivers can be saved and reused for regression testing. We expect that environment models will be changed relatively seldom unless significant errors in the requirements are discovered during testing. The environment model itself is an asset and could be reused [7].”
- AEG models can be used to gather relevant statistical data (by generating and running a large number of test scenarios) that may give insight into the effectiveness of the SUT with respect to environmental variables [8]. “From such results we can better understand which factors lead to failure of the SUT and in what way [8].”

- Environment models can be very useful to support the study of design alternatives for a SUT, especially in order to measure the efficiency of different algorithm alternatives in the SUT [7]. We have performed such experiments by subjecting each algorithm to the same scenario batches and comparing the statistical data gathered from those runs for each algorithm.

Open questions and areas that needs improvement on AEG based environment behavior models are discussed below.

- Current prototype of an automated test generator based on attributed event grammars takes an AEG model and generates a test driver in C. Since C does not support concurrency, parallel event threads represented in the AEG model (for sets, like {A, B}) are implemented by interleaving events/actions within them. However, this approach turned out to be deficient in handling and controlling the timing of events. Specifically, in our case, parallel events with false flags appeared to be taking time even though they were not executed. This kind of problem was caused by interleaved parallel events some of which had false event flags while the others had true flags. Future versions of test drivers might be implemented in a real-time programming language.
- Synchronization of two parallel events is still an open question. Specifically, in our experiment, we needed to synchronize customer events with those of shuttles' in order to make sure that customers do not send order requests once all the shuttles have retired. This was an important issue to get meaningful statistical results in our experiments and we had to adjust the life length of customer events manually.
- "In the current implementation, all loops in AEG are unfolded either using explicit iteration guards, or by assuming a random number of iterations [7]." However, this approach results in having test drivers

of great lengths (we have generated a test driver of 150,000 SLOC) putting the scalability of the approach at risk. Yet, the other option of not unfolding the loops until runtime raises another disadvantage that may increase the run time of test drivers. More experiments need to be done in order to address this problem and resolve the conflicts that may arise in either case.

LIST OF REFERENCES

[1] B. Boizer, Software Testing Techniques, Van Nostrand Reinhold, NY, 1990.

[2] B. Bruegge, A. H. Dutoit, Object Oriented Software Engineering: Using UML, Patterns and Java, Pearson Prentice Hall, NJ, 2004.

[3] P. C. Jorgensen, Software Testing: A Craftsman's Approach, CRC Press, FL, 2002.

[4] B. P. Douglass, Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns, Addison-Wesley, NJ, 2005.

[5] R. V. Binder, Testing Object Oriented Systems: Models, Patterns and Tools, Addison-Wesley, NJ, 2005.

[6] M. Auguston, J.B. Michael, M. Shing, Environment Behavior Models for Scenario Generation and Testing Automation, in: Proc. ICSE 2005 Workshop on Advances in Model-Based Software Testing, ACM, St. Louis, MO, May 2005.

[7] M. Auguston, J.B. Michael, M. Shing, Environment Behavior Models for Automation of Testing and Assessment of System Safety, Article in Press, To be Published by Elsevier B.V.

[8] M. Auguston, J.B. Michael, M. Shing, H. Tummala, D. Little, Z. Pace, Implementation and Analysis of Environment Behavior Models as a Tool for Testing Real-time, Reactive Systems, in: Proceedings of the 206 IEEE International Conference on System of Systems Engineering, Los Angeles, CA, April 2006.

[9] R. M. Hierons, H. Ural, Concerning the Ordering of Adaptive Test Sequences, in: Proc. 23rd IFIP Int. Conf. on Formal Techniques for Networked and Distributed Systems, Berlin, Germany, September 2003.

[10] Paderborn Shuttle System Case Study at http://wwwcs.upb.de/cs/ag-schaefer/CaseStudies/ShuttleSystem/. July 2006.

[11] A. Varga, OMNeT++ Discrete Simulation System (Version 3.1) User Manual, Technical University of Budapest, Dept. of Telecommunications, Hungary, March, 2005.

[12] M. Auguston, New Directions in Software Testing Automation Test Oracle Design, and Safety Assessment at http://www.nps.navy.mil/cs/auguston/Auguston-HP-RsearchDay-2005.ppt#444,5,Outlook of this presentation. July 2006.

[13] H. Robinson, Intelligent Test Automation, Software Testing & Quality Engineering Magazine, September/October 2000.

[14] M. Blackburn, R. Busser, A. Nauman, Understanding the Generations of Test Automation, Software Productivity Consortium, NFP, 2003.

[15] J. A. Whittaker, What Is Software Testing? And Why Is It So Hard?, IEEE Software, 0740-7459/00, January/February, 2000.

[16] M. Blackburn, R. Busser, A. Nauman, Why Model-Based Test Automation Is Different And What You Should Know To Get Started, Software Productivity Consortium, NFP, 2004.

[17] P. M. Maurer, Generating Test Data With Enhanced Context-Free Grammars, IEEE Software, 0740-7459/90/0700/0050, 1990.

[18] Y. Wang, D. Parnas, Simulating the Behavior of Software Modules by Trace Rewriting, IEEE Trans. Software Eng. 20(10)(1994)750-759.

[19] A. Alfonso, V. Braberman, N. Kicillof, A. Olivero, Visual Timed Event Scenarios, Proceedings of he 26th International Conference on Software Engineering, ACM Press, Edinburg, Scot., May 2004, pp.168-177.

[20] R. C. Linger, Cleanroom Software Engineering for Zero-Defect Software, IEEE, 0270-5257/93, 1993.

APPENDICES

A. OMNET++ SIMULATION MODEL CODES (C++ SOURCE FILES, C++ HEADER FILES AND OMNET++ RESOURCE FILES):

This section includes C++ source and header files for the behavioral implementation of OMNeT++ simple modules. And, OMNeT++ resource files are what define the simulation network, simple modules, module parameters and module gates used in the Paderborn simulation model.

```
*********************************************************************************
***                          C++ SOURCE FILES                                 ***
*********************************************************************************
/*********************************************************************************
Author:  Muharrem Ugur Aksu
         Naval Postgraduate School
         Computer Science Department
Date: 20 July 2006
File Name: Shuttle.cpp
*********************************************************************************/

//---------------Environment for SHUTTLE System-------------------
//----------------------------------------------------------------
#include "Shuttle.h"
Define_Module(Shuttle);

// This method is invoked when shuttles check if they have received
// messages from the Manager. The purpose of this method is to move
// the messages received from the Manager to the appropriate
// message queues.
void Shuttle::get_MANAGER_event()
{
   cMessage *m;
   while (!queue.empty())
   {
      m = (cMessage *)queue.pop();
      switch (m->kind())
      {
         case order_ev:
            order_buffer->write(((IntMsg4 *) m)->getValue1(),
                                ((IntMsg4 *) m)->getValue2(),
```

```
                        ((IntMsg4 *) m)->getValue3(),
                        ((IntMsg4 *) m)->getValue4());
            ev << "SHUTTLE received message : "
               << m->name() << ", values = "
               << ((IntMsg4 *)m)->getValue1() << " - "
               << ((IntMsg4 *)m)->getValue2() << " - "
               << ((IntMsg4 *)m)->getValue3() << " - "
               << ((IntMsg4 *)m)->getValue4() << "\n";
            break;

         case next_station_ev:
            next_station_buffer->write(((IntMsg2 *) m)->getValue1(),
                                       ((IntMsg2 *) m)->getValue2());
            ev << "SHUTTLE received message : "
               << m->name() << ", value = "
               << ((IntMsg2 *)m)->getValue1() << " - "
               << ((IntMsg2 *)m)->getValue2() <<"\n";
            break;

         case order_confirmed_ev:
            order_confirmed_buffer->write(((IntMsg6 *) m)->getValue1(),
                                          ((IntMsg6 *) m)->getValue2(),
                                          ((IntMsg6 *) m)->getValue3(),
                                          ((IntMsg6 *) m)->getValue4(),
                                          ((IntMsg6 *) m)->getValue5(),
                                          ((IntMsg6 *) m)->getValue6());

            ev << "SHUTTLE received message : "
               << m->name() << ", value = "
               << ((IntMsg6 *)m)->getValue1() << "-"
               << ((IntMsg6 *)m)->getValue2() << "-"
               << ((IntMsg6 *)m)->getValue3() << "-"
               << ((IntMsg6 *)m)->getValue4() << "-"
               << ((IntMsg6 *)m)->getValue5() << "-"
               << ((IntMsg6 *)m)->getValue6() << "\n";
            break;

         default:
            break;
      }
   }
};
```

```
// Shuttles send ready message at the beginning of the simulation
// to inform the Manager of their existence and being ready.
void Shuttle::send_ready(int shuttle_id)
{
   IntMsg *m = new IntMsg("send_ready", send_ready_ev);
   m->setValue(shuttle_id);
   ev << "SHUTTLE " << shuttle_id << " sent message : "
                   << m->name() << ", value = "
                   << ((IntMsg *)m)-> getValue() << "\n";
   send(m, "send_ready");
};


// Shuttles request an order given a shuttle id and
// order number to be requested.
void Shuttle::request_order(int shuttle_id, int order_no)
{
   IntMsg2 *m = new IntMsg2("request_order", request_order_ev);
   m->setValue1(shuttle_id);
   m->setValue2(order_no);

   ev << "SHUTTLE " << shuttle_id << " sent message : "
                   << m->name() << ", values = "
                   << ((IntMsg2 *)m)->getValue1() << " - "
                   << ((IntMsg2 *)m)->getValue2() << "\n";
   send(m, "request_order");
};


// Shuttles send their bids values together with
// their shuttle ids and order id for which the bis is made.
void Shuttle::send_bid(int shuttle_id, int bid, int order_id)
{
   IntMsg3 *m = new IntMsg3("send_bid", send_bid_ev);
   m->setValue1(shuttle_id);
   m->setValue2(bid);
   m->setValue3(order_id);

   ev << "SHUTTLE " << shuttle_id << " sent message : "
                   << m->name() << ", value = "
                   << ((IntMsg3 *)m)->getValue1() << " - "
                   << ((IntMsg3 *)m)->getValue2() << " - "
```

```
                    << ((IntMsg3 *)m)->getValue3() << "\n";
   send(m, "send_bid");
};


// Shuttles request the next station to get to the start station of an order
// given the shuttle's id and its current station.
void Shuttle::move_to_start_station(int shuttle_id, int shuttle_at_station)
{
   IntMsg2 *m = new IntMsg2("move_to_start_station", move_to_start_station_ev);
   m->setValue1(shuttle_id);
   m->setValue2(shuttle_at_station);

   ev << "SHUTTLE " << shuttle_id << " sent message : "
                    << m->name() << ", values = "
                    << ((IntMsg2 *)m)->getValue1() << " - "
                    << ((IntMsg2 *)m)->getValue2() << "\n";
   send(m, "request_next_station");
};


// Shuttles request next station enroute to destination station
// given the shuttle's id and its current station.
void Shuttle::request_next_station(int shuttle_id, int shuttle_at_station)
{
   IntMsg2 *m = new IntMsg2("request_next_station", request_next_station_ev);
   m->setValue1(shuttle_id);
   m->setValue2(shuttle_at_station);

   ev << "SHUTTLE " << shuttle_id << " sent message : "
                    << m->name() << ", values = "
                    << ((IntMsg2 *)m)->getValue1() << " - "
                    << ((IntMsg2 *)m)->getValue2() << "\n";
   send(m, "request_next_station");
};


// At the completion of an order, shuttles sends this messages to inform
// manager of their capital and retired statuses.
void Shuttle::send_order_completed(int shuttle_id, int retired, int capital)
{
   IntMsg3 *m = new IntMsg3("order_completed", order_completed_ev);
   m->setValue1(shuttle_id);
```

```
   m->setValue2(retired);
   m->setValue3(capital);

   ev << "SHUTTLE " << shuttle_id << " sent message : "
                    << m->name() << ", values = "
                    << ((IntMsg3 *)m)->getValue1() << " - "
                    << ((IntMsg3 *)m)->getValue2() << " - "
                    << ((IntMsg3 *)m)->getValue3() << "\n";
   send(m, "order_completed");
};


// Shuttles check if they have been offered an order by calling this method.
// If so it returns true and informs the shuttle of order id,
// distance of the order and number of the next order to be requested.
bool Shuttle::order(int shuttle_id, int & order_id,
                    int & route_no, int & order_request_no)
{
   get_MANAGER_event();
   if ((order_buffer->new_event()) && (shuttle_id == order_buffer->check()))
   {
      order_buffer->read1();
      order_id = order_buffer->read2();
      route_no = order_buffer->read3();
      order_request_no = order_buffer->read4();
      return true;
   }
    else
      return false;
};


// Shuttles check if they have been received next station information.
// If so it returns true and informs the shuttle of next station.
bool Shuttle::next_station(int shuttle_id, int & next_station)
{
   get_MANAGER_event();
   if ((next_station_buffer->new_event()) &&
       (shuttle_id == next_station_buffer->check()))
   {
      next_station_buffer->read1();
      next_station = next_station_buffer->read2();
      return true;
```

```
    }
   else
      return false;
};


// Shuttles check if they have received order confirmed message by calling this method.
// If so it returns true and informs the shuttle of start and destination stations of
// the order, amount of money the shuttle will be receiving upon on the completion of
// the order and number of customers assigned to the shuttle who are on the same route.
bool Shuttle::order_confirmed(int shuttle_id, int & order_confirmed,
                              int & start, int & destination,
                              int & accepted_bid, int & number_of_customers)
{
   get_MANAGER_event();
   if ((order_confirmed_buffer->new_event()) &&
       (shuttle_id == order_confirmed_buffer->check()))
   {
      order_confirmed_buffer->read1();
      order_confirmed = order_confirmed_buffer->read2();
      start = order_confirmed_buffer->read3();
      destination = order_confirmed_buffer->read4();
      accepted_bid = order_confirmed_buffer->read5();
      number_of_customers = order_confirmed_buffer->read6();
      return true;
    }
   else
   {
      return false;
   }
};


// Returns the fee incurred by shuttles by moving from one station to the next.
int Shuttle::get_transit_fee()
{
   return TRANSIT_FEE;
}


// Returns the wear incurred by shuttles by moving from one station to the next.
int Shuttle::get_transit_wear()
{
```

```
   return TRANSIT_WEAR;
}


// Returns the restored wear value for a shuttle after the completion of maintenance.
int Shuttle::get_maintenance_wear()
{
   return MAINTENANCE_WEAR;
}


// Returns the maintenance fee incurred by shuttles.
int Shuttle::get_maintenance_fee()
{
   return MAINTENANCE_FEE;
}


// Returns a unique shuttle id each time this method called.
int Shuttle::unique_id()
{
   static int unique_shuttle_id = -1;
   return ++unique_shuttle_id;
}


// Returns the default current station of all shuttles
// at the beginning of the simulation.
int Shuttle::get_shuttle_at_station()
{
   return SHUTTLE_AT_STATION;
}


// Returns the default capital values of all shuttles
// at the beginning of the simulation.
int Shuttle::get_capital()
{
   return CAPITAL;
}


// Returns the amount of money shuttles will be receiving
```

```
// for each customer except for the one for whom the shuttles
// have already made bids.
int Shuttle::get_additional_customer_toll()
{
   return ADDITIONAL_CUSTOMER_TOLL;
}


// Returns the bid amount to be offered to the Manager
// given the distance (number of links between nodes) of an order.
int Shuttle::calculate_bid(int distance)
{
   int bid = COST_OF_ONE_MOVE * distance;
   return bid;
}


// Returns the punishment fee shuttles will be incurred unless
// they do not fail to complete the order on time.
int Shuttle::get_punishment()
{
   return PUNISHMENT_FEE;
}


// ****************** CUSTOMERS ************************** //

// Customers request orders by start and destination stations.
void Shuttle::send_customer_request(int start, int dest)
{
   IntMsg2 *m = new IntMsg2("send_customer_request", send_customer_request_ev);
   m->setValue1(start);
   m->setValue2(dest);

   ev << "SHUTTLE " << " sent message : "<< m->name() << ", values = "
                    << ((IntMsg2 *)m)->getValue1() << " - "
                    << ((IntMsg2 *)m)->getValue2() << "\n";
   send(m, "route_demand");
};


// Simulates the behavior of customer order requests.
// Orders are generated randomly according to uniform distribution.
```

```
void Shuttle::get_random_request(int & start, int & dest)
{
   start = uniform(0,MAXIMUM);
   dest = uniform(0,MAXIMUM);
   while(dest == start)
      dest = uniform(0,MAXIMUM);
}

// ***************** CUSTOMERS END ************************ //

// AEG based randomly generated test drivers are included under this method.
void Shuttle::activity()
{

   order_buffer = new IntBuffer4();
   next_station_buffer = new IntBuffer2();
   order_confirmed_buffer = new IntBuffer6();

   ifstream environment_variables("EnvironmentVariables.txt");
   if(!environment_variables)
   {
      cerr << "File could not be opened" << endl;
   }
   environment_variables   >> NAMES >> TRANSIT_FEE
                           >> NAMES >> TRANSIT_WEAR
                           >> NAMES >> MAINTENANCE_WEAR
                           >> NAMES >> MAINTENANCE_FEE
                           >> NAMES >> SHUTTLE_AT_STATION
                           >> NAMES >> CAPITAL
                           >> NAMES >> ADDITIONAL_CUSTOMER_TOLL
                           >> NAMES >> COST_OF_ONE_MOVE
                           >> NAMES >> PUNISHMENT_FEE;
   environment_variables.close();
   #include "./testgenerator/s4r25.h"

   ev << "SIMULATION ENDED...";

   endSimulation();
   finish();
};

/*******************************************************************************
Author:  Muharrem Ugur Aksu
```

```
         Naval Postgraduate School
         Computer Science Department
Date: 20 July 2006
File Name: Manager.cpp
*********************************************************************************/

#include "Manager.h"

   /* THIS GRAPH IS CREATED AT RUNTIME BY GRAPH METHODS..
   bool matrix[MAXIMUM][MAXIMUM] = {
      {false,true ,false,false,false,true ,true ,false,false,false},
      {true ,false,true ,false,false,false,false,false,false,false},
      {false,true ,false,true ,false,false,true ,false,false,false},
      {false,false,true ,false,true ,false,false,false,true ,false},
      {false,false,false,true ,false,false,false,false,false,true },
      {true ,false,false,false,false,false,true ,false,false,false},
      {true ,false,true ,false,false,true ,false,true ,false,false},
      {false,false,false,false,false,false,true ,false,true ,false},
      {false,false,false,true ,false,false,false,true ,false,true },
      {false,false,false,false,true ,false,false,false,true ,false}};
   */

Define_Module(Manager);

// OMNeT++ Kernel first calls this method for initilization of variables.
// Shuttle system network is also created here by calling graph methods.

void Manager::initialize()
{
   ifstream manager_variables("ManagerVariables.txt");
   if(!manager_variables)
   {
      cerr << "File could no be opened" << endl;
   }
   manager_variables >> DUMMY >> SHUTTLE_CAPACITY;
   manager_variables.close();

   /*---------Create Directed Graph---------*/
   many_nodes = 0;
   add_nodes(10);    // Add 10 nodes (stations)..

   add_edge(0,1); add_edge(0,5); add_edge(0,6);
   add_edge(1,0); add_edge(1,2);
```

```
add_edge(2,1); add_edge(2,3); add_edge(2,6);
add_edge(3,2); add_edge(3,4); add_edge(3,8);
add_edge(4,3); add_edge(4,9);
add_edge(5,0); add_edge(5,6);
add_edge(6,0); add_edge(6,2); add_edge(6,5); add_edge(6,7);
add_edge(7,6); add_edge(7,8);
add_edge(8,3); add_edge(8,7); add_edge(8,9);
add_edge(9,4); add_edge(9,8);
/*----------------------------------------*/

// DEBUG OUTPUT: Print Directed Graph..
ev << "DIRECTED GRAPH \n";
for(int s=0; s<MAXIMUM; ++s)
{
   for(int r=0; r<MAXIMUM; ++r)
      ev << matrix[s][r] << " ";
   ev <<"\n";
}

// DEBUG OUTPUT: USE THIS TO PRINT A PATH..
/*
list<int> A = shortest_path(4,2);
while(!A.empty()) {
   ev << " " << A.front() << " -> ";
   A.pop_front();
}
*/

for(i = 0; i < MAXIMUM; ++i)
{
   shuttle_has_order[i] = false;
   shuttle_alive[i] = false;
}

for(i = 0; i < MAX_SHUTTLE_NUM; ++i)
{
   switch(i)
   {
      case 0:
         capitalVec[i].setName("Capital Vector (Shuttle 0)");
         bidConfirmedVec[i].setName(
            "Cumulative Number of Bids Confirmed (Shuttle 0)");
         bidDeniedVec[i].setName("Cumulative Number of Bids Denied (Shuttle 0)");
```

```
   capacityVec[i].setName("Number of Customers in (Shuttle 0)");
   cumulativeCapacityVec[i].setName("Cumulative Capacity of (Shuttle 0)");
   break;
case 1:
   capitalVec[i].setName("Capital Vector (Shuttle 1)");
   bidConfirmedVec[i].setName(
      "Cumulative Number of Bids Confirmed (Shuttle 1)");
   bidDeniedVec[i].setName("Cumulative Number of Bids Denied (Shuttle 1)");
   capacityVec[i].setName("Number of Customers in (Shuttle 1)");
   cumulativeCapacityVec[i].setName("Cumulative Capacity of (Shuttle 1)");
   break;
case 2:
   capitalVec[i].setName("Capital Vector (Shuttle 2)");
   bidConfirmedVec[i].setName(
      "Cumulative Number of Bids Confirmed (Shuttle 2)");
   bidDeniedVec[i].setName("Cumulative Number of Bids Denied (Shuttle 2)");
   capacityVec[i].setName("Number of Customers in (Shuttle 2)");
   cumulativeCapacityVec[i].setName("Cumulative Capacity of (Shuttle 2)");
   break;
case 3:
   capitalVec[i].setName("Capital Vector (Shuttle 3)");
   bidConfirmedVec[i].setName(
      "Cumulative Number of Bids Confirmed (Shuttle 3)");
   bidDeniedVec[i].setName("Cumulative Number of Bids Denied (Shuttle 3)");
   capacityVec[i].setName("Number of Customers in (Shuttle 3)");
   cumulativeCapacityVec[i].setName("Cumulative Capacity of (Shuttle 3)");
   break;
case 4:
   capitalVec[i].setName("Capital Vector (Shuttle 4)");
   bidConfirmedVec[i].setName(
      "Cumulative Number of Bids Confirmed (Shuttle 4)");
   bidDeniedVec[i].setName("Cumulative Number of Bids Denied (Shuttle 4)");
   capacityVec[i].setName("Number of Customers in (Shuttle 4)");
   cumulativeCapacityVec[i].setName("Cumulative Capacity of (Shuttle 4)");
   break;
case 5:
   capitalVec[i].setName("Capital Vector (Shuttle 5)");
   bidConfirmedVec[i].setName(
      "Cumulative Number of Bids Confirmed (Shuttle 5)");
   bidDeniedVec[i].setName("Cumulative Number of Bids Denied (Shuttle 5)");
   capacityVec[i].setName("Number of Customers in (Shuttle 5)");
   cumulativeCapacityVec[i].setName("Cumulative Capacity of (Shuttle 5)");
   break;
```

```
      case 6:
         capitalVec[i].setName("Capital Vector (Shuttle 6)");
         bidConfirmedVec[i].setName(
            "Cumulative Number of Bids Confirmed (Shuttle 6)");
         bidDeniedVec[i].setName("Cumulative Number of Bids Denied (Shuttle 6)");
         capacityVec[i].setName("Number of Customers in (Shuttle 6)");
         cumulativeCapacityVec[i].setName("Cumulative Capacity of (Shuttle 6)");
         break;
      case 7:
         capitalVec[i].setName("Capital Vector (Shuttle 7)");
         bidConfirmedVec[i].setName(
            "Cumulative Number of Bids Confirmed (Shuttle 7)");
         bidDeniedVec[i].setName("Cumulative Number of Bids Denied (Shuttle 7)");
         capacityVec[i].setName("Number of Customers in (Shuttle 7)");
         cumulativeCapacityVec[i].setName("Cumulative Capacity of (Shuttle 7)");
         break;
      case 8:
         capitalVec[i].setName("Capital Vector (Shuttle 8)");
         bidConfirmedVec[i].setName(
            "Cumulative Number of Bids Confirmed (Shuttle 8)");
         bidDeniedVec[i].setName("Cumulative Number of Bids Denied (Shuttle 8)");
         capacityVec[i].setName("Number of Customers in (Shuttle 8)");
         cumulativeCapacityVec[i].setName("Cumulative Capacity of (Shuttle 8)");
         break;
      case 9:
         capitalVec[i].setName("Capital Vector (Shuttle 9)");
         bidConfirmedVec[i].setName(
            "Cumulative Number of Bids Confirmed (Shuttle 9)");
         bidDeniedVec[i].setName("Cumulative Number of Bids Denied (Shuttle 9)");
         capacityVec[i].setName("Number of Customers in (Shuttle 9)");
         cumulativeCapacityVec[i].setName("Cumulative Capacity of (Shuttle 9)");
         break;
      default:
         break;
   }
   capitals[i] = 100;
   bid_confirmed[i] = 0;
   bid_denied[i] = 0;
   cumulative_shuttle_capacity[i] = 0;
   current_shuttle_station[i] = 0;
}

totalRequestVec.setName("Cumulative Number Of Requests Received");
```

```
   numberOfUnassignedOrdersVec.setName("Cumulative Number of Unassigned Orders");
   aliveShuttleVec.setName("Total Number of Alive Shuttles");
   busyShuttleVec.setName("Number of Busy Shuttles");
   processedRequestVec.setName("Cumulative Number of Orders Processed");

   timeout = new cMessage("timeout",timeout_ev);
   scheduleAt(simTime() + 10.0, timeout);

   ready_list_p = ready_list.begin();

   // common variables initialized..
   shuttle_id = 0;
   order_request_no = 0;
   shuttle_at_station = 0;
   bid = 0;
   shuttle_capital = 0;
   start, destination = -1;
   i = 0;
   alive_shuttle_counter = 0;
   total_order_request_num = 0;
   number_of_busy_shuttles = 0;
   processed_requests = 0;
   shuttle_capacity = 0;
   number_of_customers_in_queue = 0;
   order_id = 0;
};

/*----------------------------GRAPH METHODS--------------------------------*/

// Adds a node (station) to the railway network
void Manager::add_nodes()
{
   int new_node_number;
   int i;

   assert(size() < MAXIMUM);
   new_node_number = many_nodes;
   ++many_nodes;

   for(i=0; i<many_nodes; ++i)
   {
      matrix[i][new_node_number] = false;
      matrix[new_node_number][i] = false;
```

```
   }
}


// Adds a number of nodes (stations) to the railway network
void Manager::add_nodes(int number)
{
   int new_node_number;
   int i,j;

   for(j=0; j<number; ++j)
   {
      assert(size() < MAXIMUM);
      new_node_number = many_nodes;
      ++many_nodes;
      for(i=0; i<many_nodes; ++i)
      {
         matrix[i][new_node_number] = false;
         matrix[new_node_number][i] = false;
      }
   }
}


// Creates an edge (path) between given two nodes (stations)
void Manager::add_edge(int source, int target)
{
   assert(source < size());
   assert(target < size());
   matrix[source][target] = true;
}


// Removes an edge (path) between two given nodes(stations)
void Manager::remove_edge(int source, int target)
{
   assert(source < size());
   assert(target < size());
   matrix[source][target] = false;
}


// Returns true if there exists an edge (path) between two nodes (stations)
```

```
bool Manager::is_edge(int source, int target) const
{
   assert(source < size());
   assert(target < size());
   return matrix[source][target];
}


// Returns neighbors of a node (station) in a list.
list<int> Manager::neighbors(int node) const
{
   list<int> answer;
   int i;

   assert(node < size());

   for(i=0; i<size(); ++i)
   {
      if(matrix[node][i])
         answer.push_back(i);
   }
   return answer;
}


// Implements Dijkstra's Shortest Path Algorithm.
// Returns Shortest Path from Start Station s to Target Station T.
list<int> Manager::shortest_path(int s, int t)
{
   // Initialize Distance Vector to Infinity..
   // d[i] is the Distance Value from Start Station to Station i..
   int d[MAXIMUM];
   for(i=0; i<MAXIMUM; ++i)
   {
      d[i] = 100;
   }
   // Initialize Previous Station Vector to Empty..
   // p[i] is the Previous Station of Station i on the Shortest Path..
   int p[MAXIMUM];

   // S: The Settled Stations
   //    Stations Whose Shortest Distances From the Source Have Been Found..
   // Q: The Set of Unsettled Stations
```

```
// adjacent_nodes: A List of Neighbors for a Given Station..
list<int> S, Q, adjacent_nodes, Path;
list<int>::iterator S_i, Q_i, Q_erase, adjacent_nodes_i;
int u,v = -1;

Q.push_back(s);
d[s]=0;

while(!Q.empty())
{
   Q_i = Q.begin();
   int min = 100;

   // Find the Smallest (as defined by d) Station (u) in Q
   // And Remove it from Q..
   while(Q_i != Q.end())
   {
      if((d[*Q_i] < min))
      {
         min = d[*Q_i];
         Q_erase = Q_i;
      }
      Q_i++;
   }
   u = *Q_erase;
   Q.erase(Q_erase);

   // Add u to S..
   S.push_back(u);

   // Relax Neighbors of u Modifying d Vector for Each Neighbor..
   adjacent_nodes = neighbors(u);
   S_i = S.begin();
   while(S_i != S.end())
   {
      remove(adjacent_nodes.begin(), adjacent_nodes.end(),*S_i);
      S_i++;
   }
   adjacent_nodes_i = adjacent_nodes.begin();
   while(adjacent_nodes_i != adjacent_nodes.end())
   {
      v = *adjacent_nodes_i;
      if(d[v] > d[u] + 1)
```

```
         {
            d[v] = d[u] + 1;
            p[v] = u;
            Q.push_back(v);
         }
         adjacent_nodes_i++;
      }
   }

   // DEBUG OUTPUT: Print for each station the previous station on the shortest path..
   /*
   ev << "\nSHORTEST PATH FROM STATION: " << s << " TO STATION: " << t;
   ev << "\n Previous station of 9 : " << p[9];
   ev << "\n Previous station of 8 : " << p[8];
   ev << "\n Previous station of 7 : " << p[7];
   ev << "\n Previous station of 6 : " << p[6];
   ev << "\n Previous station of 5 : " << p[5];
   ev << "\n Previous station of 4 : " << p[4];
   ev << "\n Previous station of 3 : " << p[3];
   ev << "\n Previous station of 2 : " << p[2];
   ev << "\n Previous station of 1 : " << p[1];
   ev << "\n Previous station of 0 : " << p[0] << "\n";
   */

   while (p[t] >= 0)
   {
      Path.push_front(t);
      t = p[t];
   }
   Path.push_front(s);
   return Path;
}


// Implements Dijkstra's Shortest Distance Algorithm..
// Returns Shortest Distance Value from Start Station s to Target Station T..
int Manager::shortest_dist(int s, int t)
{
   // Initialize Distance Vector to Infinity..
   // d[i] is the Distance Value from Start Station to Station i..
   int d[MAXIMUM];
   for(i=0; i<MAXIMUM; ++i)
   {
```

```
   d[i] = 100;
}

// S: The Settled Stations
//    Stations Whose Shortest Distances From the Source Have Been Found..
// Q: The Set of Unsettled Stations
// adjacent_nodes: A List of Neighbors for a Given Station..
list<int> S, Q, adjacent_nodes;
list<int>::iterator S_i, Q_i, Q_erase, adjacent_nodes_i;
int u,v = -1;

Q.push_back(s);
d[s]=0;

while(!Q.empty())
{
   Q_i = Q.begin();
   int min = 100;

   // Find the Smallest (as defined by d) Station (u) in Q
   // And Remove it from Q..
   while(Q_i != Q.end())
   {
      if((d[*Q_i] < min))
      {
         min = d[*Q_i];
         Q_erase = Q_i;
      }
      Q_i++;
   }
   u = *Q_erase;
   Q.erase(Q_erase);

   // Add u to S..
   S.push_back(u);

   // Relax Neighbors of u Modifying d Vector for Each Neighbor..
   adjacent_nodes = neighbors(u);
   S_i = S.begin();
   while(S_i != S.end())
   {
      remove(adjacent_nodes.begin(), adjacent_nodes.end(),*S_i);
      S_i++;
```

```
      }
      adjacent_nodes_i = adjacent_nodes.begin();
      while(adjacent_nodes_i != adjacent_nodes.end())
      {
         v = *adjacent_nodes_i;
         if(d[v] > d[u] + 1)
         {
            d[v] = d[u] + 1;
            Q.push_back(v);
         }
         adjacent_nodes_i++;
      }
   }
   return d[t];
}
/*---------------------------GRAPH METHODS END-----------------------------*/

// Sends order information to a given shuttle.
// Parameters:
// order_id: unique order id
// distance: distance between start and destination station of the order.
// next_order_request_no:
//    (-2): no order waiting in the queue to be offered to the shuttles.
//    (-1): all the available orders have been offered,
//          ask and wait for order confirmation next time.
//    (else): next time request this order.
void Manager::send_order(int shuttle_id, int order_id,
                         int distance, int next_order_request_no)
{
   if (check_retired(shuttle_id))
   {
      IntMsg4 *m = new IntMsg4("order", order_ev);
      m->setValue1(shuttle_id);
      m->setValue2(order_id);
      m->setValue3(distance);
      m->setValue4(next_order_request_no);

      ev << "MANAGER sent message for SHUTTLE " << shuttle_id << ": "
         << m->name() << ", values = " << ((IntMsg4 *)m)->getValue2() << " - "
         << ((IntMsg4 *)m)->getValue3() << " - "
         << ((IntMsg4 *)m)->getValue4() << "\n";
      send(m, "order");
   }
```

```
};

// Sends order confirmation information to a given shuttle.
// Parameters:
// ord_confirmed: (1) to confirm, (0) to deny an offered bid.
// start: start station of the order.
// destination: final station of the order.
// accepted_bid: money the shuttle will be receiving
//               upon the completion of the order.
// number_of_customers: Number of customers assigned to the shuttle
//                      who are on the same route.
void Manager::send_order_confirmed(int shuttle_id, int ord_confirmed,
                                   int start, int destination,
                                   int accepted_bid, int number_of_customers)
{
      IntMsg6 *m = new IntMsg6("order_confirmed", order_confirmed_ev);
      m->setValue1(shuttle_id);
      m->setValue2(ord_confirmed);
      m->setValue3(start);
      m->setValue4(destination);
      m->setValue5(accepted_bid);
      m->setValue6(number_of_customers);

      ev << "MANAGER sent message : "
         << m->name() << ", value = " << ((IntMsg6 *)m)->getValue1() << " - "
                                      << ((IntMsg6 *)m)->getValue2() << " - "
                                      << ((IntMsg6 *)m)->getValue3() << " - "
                                      << ((IntMsg6 *)m)->getValue4() << " - "
                                      << ((IntMsg6 *)m)->getValue5() << " - "
                                      << ((IntMsg6 *)m)->getValue6() << "\n";
      send(m, "order_confirmed");
};


// Sends next station to be traversed to get to the start station of an order
// for a shuttle given its shuttle id and current station.
void Manager::send_next_station_to_start(int shuttle_id, int shuttle_at_station)
{
   IntMsg2 *m = new IntMsg2("next_station", next_station_ev);

   m->setValue1(shuttle_id);

   list<order_type>::iterator itr = process_list.begin();
```

```
   while (((*itr).shuttle != shuttle_id) && (itr != process_list.end()))
      itr++;
   list<int> next_station;
   if((*itr).shuttle == shuttle_id)
      next_station = shortest_path(shuttle_at_station, (*itr).start);
   m->setValue2(*(++next_station.begin()));

   current_shuttle_station[shuttle_id] = (*(++next_station.begin()));

   ev << "MANAGER sent message : "
      << m->name() << ", value = " << ((IntMsg2 *)m)->getValue1() << " -"
                                   << ((IntMsg2 *)m)->getValue2() << "\n";
   send(m, "next_station");
};


// Sends next station to be traversed
// for a shuttle given its shuttle id and current station.
void Manager::send_next_station(int shuttle_id, int shuttle_at_station)
{
   IntMsg2 *m = new IntMsg2("next_station", next_station_ev);

   m->setValue1(shuttle_id);

   list<order_type>::iterator itr = process_list.begin();
   while (((*itr).shuttle != shuttle_id) && (itr != process_list.end()))
      itr++;
   list<int> next_station;
   if((*itr).shuttle == shuttle_id)
      next_station = shortest_path(shuttle_at_station, (*itr).dest);
      m->setValue2(*(++next_station.begin()));

   current_shuttle_station[shuttle_id] = (*(++next_station.begin()));

   ev << "MANAGER sent message : "
      << m->name() << ", value = " << ((IntMsg2 *)m)->getValue1() << " - "
                           << ((IntMsg2 *)m)->getValue2() << "\n";
   send(m, "next_station");
};


// Returns false if the shuttle is not alive anymore.
int Manager::check_retired(int shuttle_id)
```

```
{
   return shuttle_alive[shuttle_id];
};


// Returns a unique order id number incremented by one each time called.
int Manager::unique_order_id()
{
   static int unique_order_id = 0;
   return ++unique_order_id;
}


// SOFTWARE CORRECTNESS TESTING - EXPERIMENT ONE
// FOUND BUG NO 1 --> ERRONEOUS ALGORITHM
/*
bool Manager::move_unique_orders()
{
   bool moved = false;
   if(process_list.empty())
   {
      moved = true;
      if(ready_list.size() > 0)
      {
         process_list.push_back(ready_list.front());
         ready_list.pop_front();
      }
   }
   if((process_list.size() < alive_shuttle_counter) &&
      (ready_list.size() > 0) && (process_list.size() > 0))
   {
      list<order_type>::iterator p_itr = process_list.begin();
      while(p_itr != process_list.end())
      {
         list<order_type>::iterator r_itr = ready_list.begin();
         while((r_itr != ready_list.end()) && (!ready_list.empty()))
         {
            path = shortest_path((*p_itr).start, (*p_itr).dest);
            sub_path = shortest_path((*r_itr).start, (*r_itr).dest);
            // If sub_path DOES NOT MATCH with or IS NOT A SUBSET of the path..
            if(path.end() == find_end(path.begin(), path.end(),
                                      sub_path.begin(), sub_path.end()))
            {
```

```
               moved = true;
               process_list.push_back(*r_itr);
               r_itr = ready_list.erase(r_itr);
               if (process_list.size() >= alive_shuttle_counter)
                  break;
            }
            else
               ++r_itr;
         }
         if (process_list.size() >= alive_shuttle_counter)
            break;
         ++p_itr;
      }
   }
   return moved;
}
*/


// Finds orders which are not on the same route
// and moves them to the process list.
bool Manager::move_unique_orders()
{
   bool moved = false;
   if(ready_list.size() > 0)
   {
      if(process_list.empty())
      {
         moved = true;
         if(ready_list.size() > 0)
         {
            process_list.push_back(ready_list.front());
            ready_list.pop_front();
         }
      }
      if((process_list.size() < alive_shuttle_counter) &&
         (ready_list.size() > 0) && (process_list.size() > 0))
      {
         list<order_type>::iterator r_itr = ready_list.begin();
         while((r_itr != ready_list.end()) && (!ready_list.empty()))
         {
            list<order_type>::iterator p_itr = process_list.begin();
            while(p_itr != process_list.end())
```

```
            {
               path = shortest_path((*p_itr).start, (*p_itr).dest);
               sub_path = shortest_path((*r_itr).start, (*r_itr).dest);
               // If sub_path DOES NOT MATCH with or IS NOT A SUBSET of the path..
               if(path.end() == find_end(path.begin(), path.end(),
                                         sub_path.begin(), sub_path.end()))
                  p_itr++;
               else
                  break;
            }
            if (p_itr == process_list.end())
            {
               moved = true;
               process_list.push_back(*r_itr);
               r_itr = ready_list.erase(r_itr);
               if (process_list.size() >= alive_shuttle_counter)
                  break;
            }
            else
               ++r_itr;
            p_itr = process_list.begin();
         }
      }
   }
   return moved;
}


// Checks if previously added orders to the process list are on the same
// route with the orders added recently to the list and if so,
// groups those orders under superset order.
bool Manager::check_duplicate_reverse()
{
   list<order_type>::iterator previous = process_list.begin();
   list<order_type>::iterator last = --(process_list.end());
   int initial_size = process_list.size();
   if(process_list.size() > 1)
   {
      while(last != process_list.begin())
      {
         while(previous != last)
         {
            if((!(*previous).locked) && ((*previous).number_of_customers +
```

```
            (*last).number_of_customers < SHUTTLE_CAPACITY))
         {
            path = shortest_path((*last).start, (*last).dest);
            sub_path = shortest_path((*previous).start, (*previous).dest);

            // If sub_path DOES NOT MATCH with or IS NOT A SUBSET of the path..
            if(path.end() == find_end(path.begin(), path.end(),
                                      sub_path.begin(), sub_path.end()))
               previous++;
            else
            {
               (*last).number_of_customers += (*previous).number_of_customers;
               // *** add sub order to the sub list & modify parent order ids ***
               if((*previous).parent_order_id == -1)
                  (*previous).parent_order_id = (*last).order_id;
               else
               {
                  (*previous).parent_order_id = (*last).order_id;
                  sub_itr = sub_list.begin();
                  while(sub_itr != sub_list.end())
                  {
                     if((*sub_itr).parent_order_id == (*previous).order_id)
                        (*sub_itr).parent_order_id = (*last).order_id;
                  }
               }
               sub_list.push_back(*previous);
               // *** ------------------------------------------------- ***
               previous = process_list.erase(previous);
            }
         }
         else
            previous++;
      }
      --last;
      previous = process_list.begin();
   }
}
if (initial_size > process_list.size())
   return true;
else
   return false;
}
```

```
// Moves the orders in ready list to process list
// in fifo fashion until process list reaches it capacity
// or ready list gets empty.
bool Manager::move_orders()
{
   bool moved = false;
   list<order_type>::iterator p_itr = process_list.begin();
   list<order_type>::iterator r_itr = ready_list.begin();
   if (process_list.size() < alive_shuttle_counter)
   {
      while((process_list.size() < alive_shuttle_counter) &&
            (!ready_list.empty()))
      {
         moved = true;
         process_list.push_back(ready_list.front());
         ready_list.pop_front();
      }
   }
   return moved;
}


// Checks process list to find ordes that might be on the same route.
// Orders on the same route are grouped under the superset order.
bool Manager::group_in_process_list()
{
   list<order_type>::iterator previous = process_list.begin();
   while((*previous).order_assigned == true)
      ++previous;
   //list<order_type>::iterator next = ++(process_list.begin());
   list<order_type>::iterator next = ++previous;
   int initial_size = process_list.size();
   if(process_list.size() > 1)
   {
      while(previous != process_list.end())
      {
         while(next != process_list.end())
         {
            if (previous == next)
               ++next;
            else if(((*previous).number_of_customers +
                     (*next).number_of_customers) < SHUTTLE_CAPACITY)
```

```
            {
               path = shortest_path((*previous).start, (*previous).dest);
               sub_path = shortest_path((*next).start, (*next).dest);
               // If sub_path DOES NOT MATCH with or IS NOT A SUBSET of the path..
               if(path.end() == find_end(path.begin(), path.end(),
                                         sub_path.begin(), sub_path.end()))
                  next++;
               else if(!(*next).locked)
               {
                  (*previous).number_of_customers += (*next).number_of_customers;
                  // *** add sub order to the sub list & modify parent order ids ***
                  if((*next).parent_order_id == -1)
                     (*next).parent_order_id = (*previous).order_id;
                  else
                  {
                     (*next).parent_order_id = (*previous).order_id;
                     sub_itr = sub_list.begin();
                     while(sub_itr != sub_list.end())
                     {
                        if((*sub_itr).parent_order_id == (*next).order_id)
                           (*sub_itr).parent_order_id = (*previous).order_id;
                     }
                  }
                  sub_list.push_back(*next);
                  // *** ------------------------------------------------------ ***
                  next = process_list.erase(next);
               }
               else
                  ++next;
            }
            else
               ++next;
         }
         ++previous;
         next = process_list.begin();
      }
   }
   if (initial_size > process_list.size())
      return true;
   else
      return false;
}
```

```
// SOFTWARE CORRECTNESS TESTING - EXPERIMENT ONE
// FOUND BUG NO 3 --> ERRONEOUS ALGORITHM
int Manager::group_matching_orders(order_type & order)
{
   int shuttle_capacity = order.number_of_customers;
   // ***         FOUND BUG NO 3       ***
   if (shuttle_capacity < SHUTTLE_CAPACITY)
   // *** ------------------------------- ***
   if (!ready_list.empty())
   {
      list<order_type>::iterator r_itr = ready_list.begin();
      path = shortest_path(order.start, order.dest);
      while(r_itr != ready_list.end())
      {
         sub_path = shortest_path((*r_itr).start, (*r_itr).dest);
         sub_path_itr = find_end(path.begin(), path.end(),
                                 sub_path.begin(), sub_path.end());
         if(sub_path_itr != (path.end()))
         {
            // *** add sub order to the sub list & modify parent order ids ***
            if((*r_itr).parent_order_id == -1)
               (*r_itr).parent_order_id = order.order_id;
            else
            {
               (*r_itr).parent_order_id = order.order_id;
               sub_itr = sub_list.begin();
               while(sub_itr != sub_list.end())
               {
                  if((*sub_itr).parent_order_id == (*r_itr).order_id)
                     (*sub_itr).parent_order_id = order.order_id;
               }
            }
            sub_list.push_back(*r_itr);
            // *** ------------------------------------------------------ ***
            r_itr = ready_list.erase(r_itr);
            processed_requests++;
            shuttle_capacity++;
            // ***         FOUND BUG NO 3    ***
            if (shuttle_capacity == SHUTTLE_CAPACITY)
               break;
            // *** --------------------------- ***
         }
```

```
         else
            ++r_itr;
      }
   }
   order.number_of_customers = shuttle_capacity;
   return shuttle_capacity;
}


// For each message received this method is called and
// appropriate action is taken according to the message type.
void Manager::handleMessage(cMessage *msg)
{
   ev << "MANAGER::handleMessage " << msg->name() <<"\n";

   switch (msg->kind()) {
      case timeout_ev:
         {
         list<order_type>::iterator itr;
         itr = ready_list.begin();
         ev << "\n***********************************";
         ev << "\nREADY LIST: ";
            while (itr != ready_list.end())
            {
               ev << (*itr).order_id << " - ";
               itr++;
            }
         itr = process_list.begin();
         ev << "\nPROCESS LIST: ";
            while (itr != process_list.end())
            {
               ev << (*itr).order_id << " - ";
               itr++;
            }
         itr = sub_list.begin();
         ev << "\nSUB LIST: ";
            while (itr != sub_list.end())
            {
               ev << (*itr).order_id << " - ";
               itr++;
            }
         ev << "\n***********************************\n";
```

```
   totalRequestVec.record(total_order_request_num);
   numberOfUnassignedOrdersVec.record(ready_list.size());
   aliveShuttleVec.record(alive_shuttle_counter);
   busyShuttleVec.record(number_of_busy_shuttles);
   processedRequestVec.record(processed_requests);

   for(i = 0; i < MAX_SHUTTLE_NUM; ++i)
   {
      capitalVec[i].record(capitals[i]);
      bidConfirmedVec[i].record(bid_confirmed[i]);
      bidDeniedVec[i].record(bid_denied[i]);
   }
   uncompletedOrdersStats.collect(ready_list.size() +
                                  sub_list.size() + process_list.size());
   scheduleAt(simTime() + 10.0, timeout);
   break;
   }

case send_ready_ev:
   shuttle_id = ((IntMsg *)msg)->getValue();
   ev << "MANAGER received message : "
      << msg->name() << ", value = " << ((IntMsg *)msg)->getValue() << "\n";
   shuttle_alive[shuttle_id] = true;
   alive_shuttle_counter++;
   break;

case request_order_ev:

   shuttle_id = ((IntMsg2 *)msg)->getValue1();
   order_request_no = ((IntMsg2 *)msg)->getValue2();

   ev << "MANAGER received message : "
      << msg->name() << ", values = " << ((IntMsg2 *)msg)->getValue1() << " - "
      << ((IntMsg2 *)msg)->getValue2() << "\n";
   /*
   //**************** MOVE ORDERS ALGORITHM 1 *****************
   algorithm_no = ALGORITHM_ONE;
   move_unique_orders();
   */

   /*
   //**************** MOVE ORDERS ALGORITHM 2 *****************
   algorithm_no = ALGORITHM_TWO;
```

```
while ((process_list.size() < alive_shuttle_counter) &&
       (ready_list.size() > 0))
{
   if (!move_unique_orders())
      break;
   if (!check_duplicate_reverse())
      break;
}
*/

/*
//**************** MOVE ORDERS ALGORITHM 3 *****************
algorithm_no = ALGORITHM_THREE;
move_orders();
*/

//**************** MOVE ORDERS ALGORITHM 4 ******************
algorithm_no = ALGORITHM_FOUR;
while ((process_list.size() < alive_shuttle_counter) &&
       (ready_list.size() > 0))
{
   if (!move_orders())
      break;
   if (!group_in_process_list())
      break;
}


/* shuttle will send "-1" to indicate that
   it has received all offered orders and
   has been requesting order confirmation.. */
if (order_request_no == -1)
{
   list<order_type>::iterator itr;
   itr = process_list.begin();
   while (((*itr).shuttle != shuttle_id) && (itr != process_list.end()))
      itr++;
   if((*itr).shuttle == shuttle_id)
   {
      number_of_busy_shuttles++;
      bid_confirmed[shuttle_id]++;
      processed_requests++;
      (*itr).order_assigned = true;
```

```
      shuttle_capacity = group_matching_orders((*itr));
      //capacityVec[shuttle_id].record((*itr).number_of_customers);
      capacityVec[shuttle_id].record(shuttle_capacity);
      cumulative_shuttle_capacity[shuttle_id] += shuttle_capacity;
      cumulativeCapacityVec[shuttle_id].record(
                    cumulative_shuttle_capacity[shuttle_id]);
      shuttleCapacityStats.collect(shuttle_capacity);

      ev << "\n\n\n CONFIRMED ORDER start end:"
         << (*itr).start << "  " << (*itr).dest << "\n";
      send_order_confirmed(shuttle_id, 1, (*itr).start, (*itr).dest,
                           (*itr).bid, (*itr).number_of_customers);

   }
   else
   {
      send_order_confirmed(shuttle_id, 0, 0, 0, 0, 0);
      bid_denied[shuttle_id]++;
   }
}

// else if requesting an order and there are available orders..
else if ((order_request_no < process_list.size()) &&
         (order_request_no < alive_shuttle_counter))
{
   if (order_request_no == 0)
   {
      process_itr = process_list.begin();

      // SOFTWARE CORRECTNESS TESTING - EXPERIMENT ONE
      // FOUND BUG NO 4 --> THE FOLLOWING PART HAS BEEN ADDED
      while ((process_itr != process_list.end()) &&
             (*process_itr).order_assigned == true)
           ++process_itr;
      // *** ------------------------------------------- ***
      if(process_itr == process_list.end())
         break;
      (*process_itr).locked = true;

      if (((order_request_no + 1) < process_list.size()) &&
          (order_request_no + 1 < alive_shuttle_counter))
      {
```

```
            send_order(shuttle_id, (*process_itr).order_id,
                               (*process_itr).dist +
                               shortest_dist(current_shuttle_station[shuttle_id],
                               (*process_itr).start),
                               ++order_request_no);
         }
         else
            send_order(shuttle_id, (*process_itr).order_id,
                               (*process_itr).dist +
                               shortest_dist(current_shuttle_station[shuttle_id],
                               (*process_itr).start), -1);
      }
      else
      { // if (order_request_no == 1 or more)

         process_itr = process_list.begin();
         for (i=0; i < order_request_no; i++)
            process_itr++;
         (*process_itr).locked = true;

         if (((order_request_no + 1) < process_list.size()) &&
             (order_request_no + 1 < alive_shuttle_counter))
            send_order(shuttle_id, (*process_itr).order_id,
                               (*process_itr).dist +
                               shortest_dist(current_shuttle_station[shuttle_id],
                               (*process_itr).start),
                               ++order_request_no);
         else
            send_order(shuttle_id, (*process_itr).order_id,
                               (*process_itr).dist +
                               shortest_dist(current_shuttle_station[shuttle_id],
                               (*process_itr).start), -1);
      }
   }

   /* no available orders at the moment..
      shuttle(s) should wait until some orders become available.. */
   else
      send_order(shuttle_id, 0, 0, -2);
   break;

case send_bid_ev:
```

```
// SOFTWARE CORRECTNESS TESTING - EXPERIMENT ONE
// FOUND BUG NO 2 --> CHANGE UNIQUE ORDER IDENTIFICATION
/*
shuttle_id = ((IntMsg4 *)msg)->getValue1();
bid = ((IntMsg4 *)msg)->getValue2();
start = ((IntMsg4 *)msg)->getValue3();
destination = ((IntMsg4 *)msg)->getValue4();

ev << "MANAGER received message : "
   << msg->name() << ", value = " << ((IntMsg4 *)msg)->getValue1() << " - "
                                  << ((IntMsg4 *)msg)->getValue2() << " - "
                                  << ((IntMsg4 *)msg)->getValue3() << " - "
                                  << ((IntMsg4 *)msg)->getValue4() << "\n";

   itr = process_list.begin();
   for(i=0; i<alive_shuttle_counter; ++i)
   {
      if (((*itr).start != start) || ((*itr).dest != destination))
         itr++;
   }
*/

shuttle_id = ((IntMsg3 *)msg)->getValue1();
bid = ((IntMsg3 *)msg)->getValue2();
order_id = ((IntMsg3 *)msg)->getValue3();

ev << "MANAGER received message : "
   << msg->name() << ", value = " << ((IntMsg3 *)msg)->getValue1() << " - "
                                  << ((IntMsg3 *)msg)->getValue2() << " - "
                                  << ((IntMsg3 *)msg)->getValue3() << "\n";

   itr = process_list.begin();

   for(i=0; i<alive_shuttle_counter; ++i)
   {
      if (((*itr).order_id != order_id))
         itr++;
   }

   assert((*itr).order_id == order_id);

   ev << "\n\n  " << (*itr).order_id << "  " << order_id << " \n";
```

```
      if (!shuttle_has_order[shuttle_id])
      {
         if (bid < (*itr).bid) {
            shuttle_has_order[(*itr).shuttle] = false;
            (*itr).bid = bid;
            (*itr).shuttle = shuttle_id;
            shuttle_has_order[shuttle_id] = true;
         }
      }
   break;

case move_to_start_station_ev:
   shuttle_id = ((IntMsg2 *)msg)->getValue1();
   shuttle_at_station = ((IntMsg2 *)msg)->getValue2();

   ev << "MANAGER received message : "
      << msg->name() << ", values = " << ((IntMsg2 *)msg)->getValue1() << " - "
                                      << ((IntMsg2 *)msg)->getValue2() << "\n";

   send_next_station_to_start(shuttle_id, shuttle_at_station);
   break;

case request_next_station_ev:

   shuttle_id = ((IntMsg2 *)msg)->getValue1();
   shuttle_at_station = ((IntMsg2 *)msg)->getValue2();

   ev << "MANAGER received message : "
      << msg->name() << ", values = " << ((IntMsg2 *)msg)->getValue1() << " - "
                                      << ((IntMsg2 *)msg)->getValue2() << "\n";

   // *** collect waiting time stats for primary orders ***
   itr = process_list.begin();
   while (((*itr).shuttle != shuttle_id) && (itr != process_list.end()))
         itr++;
   if((*itr).shuttle == shuttle_id)
      if((*itr).start == shuttle_at_station)
      {
         (*itr).wait_time = simTime() - (*itr).wait_time;
         customerWaitTimeStats.collect((*itr).wait_time);
      }
   // *** ------------------------------------------- ***
```

```
   // *** collect waiting time stats for sub orders ***
   sub_itr = sub_list.begin();
   while(sub_itr != sub_list.end())
   {
      if((*sub_itr).parent_order_id == (*itr).order_id)
      {
         if((*sub_itr).start == shuttle_at_station)
         {
            (*sub_itr).wait_time = simTime() - (*sub_itr).wait_time;
            customerWaitTimeStats.collect((*sub_itr).wait_time);
            sub_itr = sub_list.erase(sub_itr);
         }
         else
            ++sub_itr;
      }
      else
         ++sub_itr;
   }
   // *** ------------------------------------------- ***

   send_next_station(shuttle_id, shuttle_at_station);
   break;

case order_completed_ev:

   ev << "MANAGER received message : "<< msg->name() << ", values = "
      << ((IntMsg3 *)msg)->getValue1() << " - "
      << ((IntMsg3 *)msg)->getValue2() << " - "
      << ((IntMsg3 *)msg)->getValue3() << "\n";

   shuttle_id = ((IntMsg3 *)msg)->getValue1();
   shuttle_alive[shuttle_id] = ((IntMsg3 *)msg)->getValue2();
   shuttle_capital = ((IntMsg3 *)msg)->getValue3();

   shuttleCapitalStats.collect(shuttle_capital);

   shuttle_has_order[shuttle_id] = false;

   capitals[shuttle_id] = shuttle_capital;

   number_of_busy_shuttles--;
```

```
itr = process_list.begin();
while ((*itr).shuttle != shuttle_id)
   itr++;
   if(itr != process_list.end())
      ev << "\n DEBUG THIS ERROR \n";
if((*itr).shuttle == shuttle_id)
{
   ev << "\n Deleting : " << (*itr).order_id << "\n";
   process_list.erase(itr);
}

/*
//**************** MOVE ORDERS ALGORITHM 1 *****************
if (!(process_list.size() >= alive_shuttle_counter))
   move_unique_orders();
*/

/*
//**************** MOVE ORDERS ALGORITHM 2 *****************
if (!(process_list.size() >= alive_shuttle_counter))
{
   while ((process_list.size() < alive_shuttle_counter) &&
          (ready_list.size() > 0))
   {
      if (!move_unique_orders())
         break;
      if (!check_duplicate_reverse())
         break;
   }
}
*/

/*
//**************** MOVE ORDERS ALGORITHM 3 *****************
if (!(process_list.size() >= alive_shuttle_counter))
   move_orders();
*/

//**************** MOVE ORDERS ALGORITHM 4 *****************
if (!(process_list.size() >= alive_shuttle_counter))
{
   while ((process_list.size() < alive_shuttle_counter) &&
                                 (ready_list.size() > 0))
```

```
            {
               if (!move_orders())
                  break;
               if (!group_in_process_list())
                  break;
            }
         }

         if (!shuttle_alive[shuttle_id]) {
            alive_shuttle_counter--;
            ev << "----------> SHUTTLE " << shuttle_id << " RETIRED !!\n";
         }
         break;

      case send_customer_request_ev:
         {

         ev << "MANAGER received message : "
            << msg->name() << ", values = " << ((IntMsg2 *)msg)->getValue1() << " - "
                           << ((IntMsg2 *)msg)->getValue2() << "\n ";
         order_type order;;
         order.start = ((IntMsg2 *)msg)->getValue1();
         order.dest = ((IntMsg2 *)msg)->getValue2();
         order.dist = shortest_dist(order.start, order.dest);
         order.bid = 1000;    // Dummy Value..
         order.shuttle = -1;     // Dummy Value..
         order.number_of_customers = 1;
         order.wait_time = simTime();
         order.order_id = unique_order_id();
         order.parent_order_id = -1;
         order.locked = false;
         order.order_assigned = false;
         ready_list.push_back(order);
         total_order_request_num++;

         break;
         }

      default:
         ev << "MANAGER: Error in HandleMessage - Unexpected Message Kind...\n";
         break;
   }
};
```

```
// This method is called by the simulation kernel
// at the end of the simulation and
// simulation results are recorded.
void Manager::finish()
{

   ofstream experiment_stats;
   switch(algorithm_no)
   {
   case ALGORITHM_ONE:
      experiment_stats.open("AlgorithmStatsOne.txt", ios::app);
      break;
   case ALGORITHM_TWO:
      experiment_stats.open("AlgorithmStatsTwo.txt", ios::app);
      break;
   case ALGORITHM_THREE:
      experiment_stats.open("AlgorithmStatsThree.txt", ios::app);
      break;
   case ALGORITHM_FOUR:
      experiment_stats.open("AlgorithmStatsFour.txt", ios::app);
      break;
   default:
      experiment_stats.open("Error.txt", ios::app);
      break;
   }
   if(!experiment_stats)
   {
      cerr << "File could not be opened" << endl;
   }

   ev <<"\nMin of Uncompleted Orders/10 sec : " << uncompletedOrdersStats.min();
   ev <<"\nMax of Uncompleted Orders/10 sec : " << uncompletedOrdersStats.max();
   ev <<"\nMean of Uncompleted Orders/10 sec : " << uncompletedOrdersStats.mean();
   experiment_stats << uncompletedOrdersStats.mean() << " ";
   ev <<"\nStandard Deviation of Uncompleted Orders/10 sec : "
      << uncompletedOrdersStats.stddev();
   ev <<"\n--------------------------------------------------------------------------";
   ev <<"\nMin of Shuttle Capacity/Order : " << shuttleCapacityStats.min();
   ev <<"\nMax of Shuttle Capacity/Order : " << shuttleCapacityStats.max();
   ev <<"\nMean of Shuttle Capacity/Order : " << shuttleCapacityStats.mean();
   experiment_stats << shuttleCapacityStats.mean() << " ";
```

```
   ev <<"\nStandard Deviation of Shuttle Capacity/Order : "
      << shuttleCapacityStats.stddev();
   ev <<"\n--------------------------------------------------------------------------";
   ev <<"\nMin of Shuttle Capital/Order : " << shuttleCapitalStats.min();
   experiment_stats << shuttleCapitalStats.min() << " ";
   ev <<"\nMax of Shuttle Capital/Order : " << shuttleCapitalStats.max();
   ev <<"\nMean of Shuttle Capital/Order : " << shuttleCapitalStats.mean();
   experiment_stats << shuttleCapitalStats.mean() << " ";
   ev <<"\nStandard Deviation of Shuttle Capital/ Order : "
      << shuttleCapitalStats.stddev();
   ev <<"\n--------------------------------------------------------------------------";
   ev <<"\nMin of Customer Wait Time : " << simtimeToStr(customerWaitTimeStats.min());
   experiment_stats << customerWaitTimeStats.min() << " ";
   ev <<"\nMax of Customer Wait Time : " << simtimeToStr(customerWaitTimeStats.max());
   experiment_stats << customerWaitTimeStats.max() << " ";
   ev <<"\nMean of Customer Wait Time : " << simtimeToStr(customerWaitTimeStats.mean());
   experiment_stats << customerWaitTimeStats.mean() << endl;
   ev <<"\nStandard Deviation Customer Wait Time : "
      << simtimeToStr(customerWaitTimeStats.stddev());
   experiment_stats.close();

   uncompletedOrdersStats.recordScalar("Waiting Requests..");
   shuttleCapacityStats.recordScalar("Shuttle Capacity..");
   shuttleCapitalStats.recordScalar("Shuttle Capital..");
   customerWaitTimeStats.recordScalar("Customer Wait Times..");
};

/*****************************************************************************
Author:  Muharrem Ugur Aksu
         Naval Postgraduate School
         Computer Science Department
Date: 20 July 2006
File Name: intbuffer.cpp
*****************************************************************************/

#include "intbuffer.h"

IntBuffer::IntBuffer()
{
   this->new_event_flag = false;
}

int IntBuffer::check()
```

```
{
   int return_val = value_var.front();
   return (return_val);
}

int IntBuffer::read()
{
   int return_val = value_var.front();
   value_var.pop_front();
   if (value_var.empty())
      this->new_event_flag = false;
   return (return_val);
}

void IntBuffer::write(int x)
{
   this->value_var.push_back(x);
   this->new_event_flag = true;
};

bool IntBuffer::new_event()
{
   return (this->new_event_flag);
}

/*****************************************************************************
Author:  Muharrem Ugur Aksu
         Naval Postgraduate School
         Computer Science Department
Date: 20 July 2006
File Name: intbuffer2.cpp
*****************************************************************************/

#include "intbuffer2.h"

IntBuffer2::IntBuffer2()
{
   this->new_event_flag = false;
}

int IntBuffer2::check()
{
   int return_val = value_var.front();
```

```
   return (return_val);
}

int IntBuffer2::read1()
{
   int return_val = value_var.front();
   value_var.pop_front();
   return (return_val);
}

int IntBuffer2::read2()
{
   int return_val = value_var.front();
   value_var.pop_front();
   if (value_var.empty())
      this ->new_event_flag = false;
   return (return_val);
}

void IntBuffer2::write(int x, int y)
{
   this->value_var.push_back(x);
   this->value_var.push_back(y);
   this->new_event_flag = true;
};

bool IntBuffer2::new_event()
{
   return (this->new_event_flag);
}

/*****************************************************************************
Author:  Muharrem Ugur Aksu
         Naval Postgraduate School
         Computer Science Department
Date: 20 July 2006
File Name: intbuffer3.cpp
*****************************************************************************/

#include "intbuffer3.h"

IntBuffer3::IntBuffer3()
{
```

```
   this->new_event_flag = false;
}

int IntBuffer3::check()
{
   int return_val = value_var.front();
   return (return_val);
}

int IntBuffer3::read1()
{
   int return_val = value_var.front();
   value_var.pop_front();
   return (return_val);
}

int IntBuffer3::read2()
{
   int return_val = value_var.front();
   value_var.pop_front();
   return (return_val);
}

int IntBuffer3::read3()
{
   int return_val = value_var.front();
   value_var.pop_front();
   if (value_var.empty())
      this->new_event_flag = false;
   return (return_val);
}

void IntBuffer3::write(int x, int y, int z)
{
   this->value_var.push_back(x);
   this->value_var.push_back(y);
   this->value_var.push_back(z);

   this->new_event_flag = true;
};

bool IntBuffer3::new_event()
{
```

```
   return (this->new_event_flag);
}

/*****************************************************************************************
Author:  Muharrem Ugur Aksu
         Naval Postgraduate School
         Computer Science Department
Date: 20 July 2006
File Name: intbuffer4.cpp
*****************************************************************************************/

#include "intbuffer4.h"

IntBuffer4::IntBuffer4()
{
   this->new_event_flag = false;
}

int IntBuffer4::check()
{
   int return_val = value_var.front();
   return (return_val);
}

int IntBuffer4::read1()
{
   int return_val = value_var.front();
   value_var.pop_front();
   return (return_val);
}

int IntBuffer4::read2()
{
   int return_val = value_var.front();
   value_var.pop_front();
   return (return_val);
}

int IntBuffer4::read3()
{
   int return_val = value_var.front();
   value_var.pop_front();
   return (return_val);
```

```
}

int IntBuffer4::read4()
{
   int return_val = value_var.front();
   value_var.pop_front();
   if (value_var.empty())
      this->new_event_flag = false;
   return (return_val);
}

void IntBuffer4::write(int x, int y, int z, int m)
{
   this->value_var.push_back(x);
   this->value_var.push_back(y);
   this->value_var.push_back(z);
   this->value_var.push_back(m);

   this->new_event_flag = true;
};

bool IntBuffer4::new_event()
{
   return (this->new_event_flag);
}

/*********************************************************************************************
Author:  Muharrem Ugur Aksu
         Naval Postgraduate School
         Computer Science Department
Date: 20 July 2006
File Name: intbuffer5.cpp
*********************************************************************************************/

#include "intbuffer5.h"

IntBuffer5::IntBuffer5()
{
   this->new_event_flag = false;
}

int IntBuffer5::check()
{
```

```
   int return_val = value_var.front();
   return (return_val);
}

int IntBuffer5::read1()
{
   int return_val = value_var.front();
   value_var.pop_front();
   return (return_val);
}

int IntBuffer5::read2()
{
   int return_val = value_var.front();
   value_var.pop_front();
   return (return_val);
}

int IntBuffer5::read3()
{
   int return_val = value_var.front();
   value_var.pop_front();
   return (return_val);
}

int IntBuffer5::read4()
{
   int return_val = value_var.front();
   value_var.pop_front();
   return (return_val);
}

int IntBuffer5::read5()
{
   int return_val = value_var.front();
   value_var.pop_front();
   if (value_var.empty())
      this->new_event_flag = false;
   return (return_val);
}

void IntBuffer5::write(int x, int y, int z, int m, int n)
{
```

```
   this->value_var.push_back(x);
   this->value_var.push_back(y);
   this->value_var.push_back(z);
   this->value_var.push_back(m);
   this->value_var.push_back(n);

   this->new_event_flag = true;
};

bool IntBuffer5::new_event()
{
   return (this->new_event_flag);
}

/*****************************************************************************************
Author:  Muharrem Ugur Aksu
         Naval Postgraduate School
         Computer Science Department
Date: 20 July 2006
File Name: intbuffer6.cpp
*****************************************************************************************/

#include "intbuffer6.h"

IntBuffer6::IntBuffer6()
{
   this->new_event_flag = false;
}

int IntBuffer6::check()
{
   int return_val = value_var.front();
   return (return_val);
}

int IntBuffer6::read1()
{
   int return_val = value_var.front();
   value_var.pop_front();
   return (return_val);
}

int IntBuffer6::read2()
```

```
{
   int return_val = value_var.front();
   value_var.pop_front();
   return (return_val);
}

int IntBuffer6::read3()
{
   int return_val = value_var.front();
   value_var.pop_front();
   return (return_val);
}

int IntBuffer6::read4()
{
   int return_val = value_var.front();
   value_var.pop_front();
   return (return_val);
}

int IntBuffer6::read5()
{
   int return_val = value_var.front();
   value_var.pop_front();
   return (return_val);
}

int IntBuffer6::read6()
{
   int return_val = value_var.front();
   value_var.pop_front();
   if (value_var.empty())
      this->new_event_flag = false;
   return (return_val);
}

void IntBuffer6::write(int x, int y, int z, int m, int n, int o)
{
   this->value_var.push_back(x);
   this->value_var.push_back(y);
   this->value_var.push_back(z);
   this->value_var.push_back(m);
   this->value_var.push_back(n);
```

```
   this->value_var.push_back(o);

   this->new_event_flag = true;
};

bool IntBuffer6::new_event()
{
   return (this->new_event_flag);
}

//*************************************************************************************
//***                          C++ HEADER FILES                                     ***
//*************************************************************************************
/*************************************************************************************
Author:  Muharrem Ugur Aksu
         Naval Postgraduate School
         Computer Science Department
Date: 20 July 2006
File Name: EnvironmentParameters.h
*************************************************************************************/

#ifndef __ENVIRONMENT_PARAMETERS_H
#define __ENVIRONMENT_PARAMETERS_H

const int TRANSIT_FEE = 2;
const int TRANSIT_WEAR = 1;
const int MAINTENANCE_WEAR = 10;
const int MAINTENANCE_FEE = 10;
const int SHUTTLE_AT_STATION = 1;
const int CAPITAL = 100;
const int ADDITIONAL_CUSTOMER_TOLL = 2;
const int COST_OF_ONE_MOVE = (MAINTENANCE_FEE /
                              (MAINTENANCE_WEAR / TRANSIT_WEAR))
                               + TRANSIT_FEE;
const int PUNISHMENT_FEE = 4;

#endif

/*************************************************************************************
Author:  Muharrem Ugur Aksu
         Naval Postgraduate School
         Computer Science Department
Date: 20 July 2006
```

```
File Name: ManagerParameters.h
*****************************************************************************************/

#ifndef __MANAGER_PARAMETERS_H
#define __MANAGER_PAREMETERS_H

const int MAXIMUM = 10; // MAXIMUM NUMBER OF NODES IN THE GRAPH..
const int MAX_SHUTTLE_NUM = 10;

#endif __MANAGER_PARAMETERS_H

/*****************************************************************************************
Author:  Muharrem Ugur Aksu
         Naval Postgraduate School
         Computer Science Department
Date: 20 July 2006
File Name: event_kind.h
*****************************************************************************************/

#ifndef _EVENT_KIND_H_
#define _EVENT_KIND_H_

#define send_ready_ev               1
#define request_order_ev            2
#define move_to_start_station_ev    3
#define request_next_station_ev     4
#define order_ev                    5
#define next_station_ev             6
#define order_completed_ev          7
#define send_bid_ev                 8
#define order_confirmed_ev          9
#define timeout_ev                  10
#define send_customer_request_ev    11
#endif

/*****************************************************************************************
Author:  Muharrem Ugur Aksu
         Naval Postgraduate School
         Computer Science Department
```

```
Date: 20 July 2006
File Name: intbuffer.h
*******************************************************************************/

#ifndef _INTBUFFER_H_
#define _INTBUFFER_H_
#include <list>
using namespace std;

class IntBuffer
{
  protected:
   list<int> value_var;
   bool new_event_flag;
  public:
    IntBuffer();
   int check();
   int read();
   void write(int x);
   bool new_event();
};
#endif

/*******************************************************************************
Author:  Muharrem Ugur Aksu
         Naval Postgraduate School
         Computer Science Department
Date: 20 July 2006
File Name: intbuffer2.h
*******************************************************************************/

#ifndef _INTBUFFER2_H_
#define _INTBUFFER2_H_
#include <list>
using namespace std;

class IntBuffer2
{
  protected:
    //int value_var[2];
   list<int> value_var;
   bool new_event_flag;
```

```
  public:
    IntBuffer2();
   int check();
   int read1();
   int read2();
   void write(int x,int y);
   bool new_event();
};
#endif

/*********************************************************************************************
Author:  Muharrem Ugur Aksu
         Naval Postgraduate School
         Computer Science Department
Date: 20 July 2006
File Name: intbuffer3.h
*********************************************************************************************/

#ifndef _INTBUFFER3_H_
#define _INTBUFFER3_H_
#include <list>
using namespace std;

class IntBuffer3
{
  protected:
   list<int> value_var;
   bool new_event_flag;
  public:
    IntBuffer3();
   int check();
   int read1();
   int read2();
   int read3();
   void write(int x, int y, int z);
   bool new_event();
};
#endif

/*********************************************************************************************
Author:  Muharrem Ugur Aksu
         Naval Postgraduate School
         Computer Science Department
```

```
Date: 20 July 2006
File Name: intbuffer4.h
*******************************************************************************/

#ifndef _INTBUFFER4_H_
#define _INTBUFFER4_H_
#include <list>
using namespace std;

class IntBuffer4
{
  protected:
   list<int> value_var;
   bool new_event_flag;
  public:
    IntBuffer4();
   int check();
   int read1();
   int read2();
   int read3();
   int read4();
   void write(int x, int y, int z, int m);
   bool new_event();
};
#endif

/*******************************************************************************
Author:  Muharrem Ugur Aksu
         Naval Postgraduate School
         Computer Science Department
Date: 20 July 2006
File Name: intbuffer5.h
*******************************************************************************/

#ifndef _INTBUFFER5_H_
#define _INTBUFFER5_H_
#include <list>
using namespace std;

class IntBuffer5
{
  protected:
   list<int> value_var;
```

```
   bool new_event_flag;
  public:
   IntBuffer5();
   int check();
   int read1();
   int read2();
   int read3();
   int read4();
   int read5();
   void write(int x, int y, int z, int m, int n);
   bool new_event();
};
#endif

/*********************************************************************************************
Author:  Muharrem Ugur Aksu
         Naval Postgraduate School
         Computer Science Department
Date: 20 July 2006
File Name: intbuffer6.h
*********************************************************************************************/

#ifndef _INTBUFFER6_H_
#define _INTBUFFER6_H_
#include <list>
using namespace std;

class IntBuffer6
{
  protected:
   list<int> value_var;
   bool new_event_flag;
  public:
    IntBuffer6();
   int check();
   int read1();
   int read2();
   int read3();
   int read4();
   int read5();
   int read6();
   void write(int x, int y, int z, int m, int n, int o);
   bool new_event();
```

```
};
#endif

/*******************************************************************************************
Author:  Muharrem Ugur Aksu
         Naval Postgraduate School
         Computer Science Department
Date: 20 July 2006
File Name: Manager.h
*******************************************************************************************/

#ifndef __MANAGER_H
#define __MANAGER_H

#include <omnetpp.h>
#include "intbuffer.h"
#include "intbuffer2.h"
#include "event_kind.h"
#include "intmsg_m.h"
#include "intmsg2_m.h"
#include "intmsg3_m.h"
#include "intmsg4_m.h"
#include "intmsg5_m.h"
#include "intmsg6_m.h"

#include <list>
#include <cassert>
#include <algorithm>
#include <iostream>
#include <fstream>
#include <cstdlib>

using std::ifstream;
using std::ofstream;
using std::fstream;
using std::endl;
using std::cerr;
#include "ManagerParameters.h"
using namespace std;

enum algorithm_number {ALGORITHM_ONE = 1, ALGORITHM_TWO,
                       ALGORITHM_THREE, ALGORITHM_FOUR} algorithm_no;
```

```
class Manager : public cSimpleModule
{
  protected:

   cOutVector capitalVec[MAX_SHUTTLE_NUM];
   cOutVector bidConfirmedVec[MAX_SHUTTLE_NUM];
   cOutVector bidDeniedVec[MAX_SHUTTLE_NUM];
   cOutVector capacityVec[MAX_SHUTTLE_NUM];
   cOutVector cumulativeCapacityVec[MAX_SHUTTLE_NUM];
   cOutVector totalRequestVec;
   cOutVector numberOfUnassignedOrdersVec;
   cOutVector processedRequestVec;
   cOutVector aliveShuttleVec;
   cOutVector busyShuttleVec;

   cLongHistogram uncompletedOrdersStats;
   cLongHistogram shuttleCapacityStats;
   cLongHistogram shuttleCapitalStats;
   cLongHistogram customerWaitTimeStats;

  public:

   class order_type {
      public:
      int start;
      int dest;
      int dist;
      int bid;
      int shuttle;
      int number_of_customers;
      int order_id;
      int parent_order_id;
      bool locked;
      bool order_assigned;
      simtime_t wait_time;
   };

   cMessage *m;
   cMessage *order;
   cMessage *next_station;
   cMessage *timeout;

   /*--------------------GRAPH METHODS-----------------------*/
```

```
 void add_nodes();
 void add_nodes(int number);
 void add_edge(int source, int target);
 void remove_edge(int source, int target);
 int size() const { return many_nodes; }
 bool is_edge(int source, int target) const;
 list<int> neighbors(int node) const;

 list<int> shortest_path(int s, int t);
 int shortest_dist(int s, int t);
 /*-------------------------------------------------------*/

 Module_Class_Members(Manager, cSimpleModule, 0);

 virtual void initialize();
  virtual void handleMessage(cMessage *msg);
 virtual void finish();

 void send_order(int shuttle_id, int order_id,
                 int distance, int next_order_request_no);
 void send_next_station(int shuttle_id, int shuttle_at_station);
 void send_next_station_to_start(int shuttle_id, int shuttle_at_station);
 void send_order_confirmed(int shuttle_id, int ord_confirmed, int start,
                           int destination, int accepted_bid,
                           int number_of_customers);
 void move_unique_order();

 int check_retired(int shuttle_id);
 int group_matching_orders(order_type & order);
 int unique_order_id();

 bool move_unique_orders();
 bool check_duplicate_reverse();
 bool move_orders();
 bool group_in_process_list();

private:

 simtime_t simulation_time;

 list<order_type> ready_list;
 list<order_type>::iterator ready_list_p;
 list<order_type> process_list;
```

```
   list<order_type>::iterator process_itr;;
   list<order_type>::iterator itr;
   list<order_type> sub_list;
   list<order_type>::iterator sub_itr;;

   list<int> path ,sub_path;
   list<int>::iterator sub_path_itr;

   int shuttle_id;
   int shuttle_at_station;
   int shuttle_capital;
   int bid;
   int start;
   int destination;
   int order_request_no;
   int alive_shuttle_counter;
   int number_of_customers_in_queue;
   int order_id;
   // Number of Nodes in the Graph..
   int many_nodes;
   int i;
   int total_order_request_num;
   int capitals[MAX_SHUTTLE_NUM];
   int bid_confirmed[MAX_SHUTTLE_NUM];
   int bid_denied[MAX_SHUTTLE_NUM];
   int cumulative_shuttle_capacity[MAX_SHUTTLE_NUM];
   int number_of_busy_shuttles;
   int processed_requests;
   int shuttle_capacity;
   int current_shuttle_station[MAX_SHUTTLE_NUM];
   int SHUTTLE_CAPACITY;

   bool shuttle_has_order[MAX_SHUTTLE_NUM];
   bool shuttle_alive[MAX_SHUTTLE_NUM];
   // Adjacency Matrix for the Graph Implementation..
   bool matrix[MAXIMUM][MAXIMUM];

   char DUMMY[50];
};
#endif __MANAGER_H

/*****************************************************************************
Author:  Muharrem Ugur Aksu
```

```
         Naval Postgraduate School
         Computer Science Department
Date: 20 July 2006
File Name: Shuttle.h
*********************************************************************************/

#ifndef __SHUTTLE_H
#define __SHUTTLE_H

#include <omnetpp.h>
#include "intbuffer.h"
#include "intbuffer2.h"
#include "intbuffer3.h"
#include "intbuffer4.h"
#include "intbuffer5.h"
#include "intbuffer6.h"
#include "intmsg_m.h"
#include "intmsg2_m.h"
#include "intmsg3_m.h"
#include "intmsg4_m.h"
#include "intmsg5_m.h"
#include "intmsg6_m.h"
#include "event_kind.h"
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string>

using std::ifstream;
using std::endl;
using std::cerr;
using std::string;

const int MAXIMUM = 10; // MAXIMUM NUMBER OF NODES IN THE GRAPH..

class Shuttle : public cSimpleModule
{

public:
   Module_Class_Members(Shuttle, cSimpleModule, 16384);

   cQueue queue;
   IntBuffer4 *order_buffer;
```

```
   IntBuffer2 *next_station_buffer;
   IntBuffer6 *order_confirmed_buffer;

   virtual void activity();
   void get_MANAGER_event();
   void send_ready(int shuttle_id);
   void request_order(int shuttle_id, int order_no);
   void move_to_start_station(int shuttle_id, int shuttle_at_station);
   void request_next_station(int shuttle_id, int shuttle_at_station);
   void send_order_completed(int shuttle_id, int retired, int capital);
   void send_bid(int shuttle_id, int bid, int order_id);

   bool order(int shuttle_id, int & order_id,
              int & route_no, int & order_request_no);
   bool next_station(int shuttle_id, int & next_station);
   bool order_confirmed(int shuttle_id, int & ord_confirmed,
                        int & start, int & destination,
                        int & accepted_bid, int & number_of_customers);

   int get_transit_fee();
   int get_transit_wear();
   int get_maintenance_wear();
   int get_maintenance_fee();
   int get_additional_customer_toll();
   int get_punishment();
   int unique_id();
   int get_shuttle_at_station();
   int get_capital();
   int calculate_bid(int distance);

   // CUSTOMERS

   void get_random_request(int & start, int & dest);
   void send_customer_request(int start, int dest);
private:
   int TRANSIT_FEE;
   int TRANSIT_WEAR;
   int MAINTENANCE_WEAR;
   int MAINTENANCE_FEE;
   int SHUTTLE_AT_STATION;
   int CAPITAL;
   int ADDITIONAL_CUSTOMER_TOLL;
   int COST_OF_ONE_MOVE;
```

```
   int PUNISHMENT_FEE;
   char NAMES[50];
};

#endif

//*************************************************************************************
//***                          OMNeT++ RESOURCE FILES                               ***
//*************************************************************************************
/*************************************************************************************
Author:  Muharrem Ugur Aksu
         Naval Postgraduate School
         Computer Science Department
Date: 20 July 2006
File Name: intmsg.msg
*************************************************************************************/

message IntMsg
{
    fields:
        int value;
};

/*************************************************************************************
Author:  Muharrem Ugur Aksu
         Naval Postgraduate School
         Computer Science Department
Date: 20 July 2006
File Name: intmsg2.msg
*************************************************************************************/

message IntMsg2
{
    fields:
      int value1;
      int value2;
};

/*************************************************************************************
Author:  Muharrem Ugur Aksu
         Naval Postgraduate School
         Computer Science Department
Date: 20 July 2006
```

```
File Name: intmsg3.msg
*******************************************************************************************/

message IntMsg3
{
    fields:
      int value1;
      int value2;
      int value3;
};

/*******************************************************************************************
Author:  Muharrem Ugur Aksu
         Naval Postgraduate School
         Computer Science Department
Date: 20 July 2006
File Name: intmsg4.msg
*******************************************************************************************/

message IntMsg4
{
    fields:
      int value1;
      int value2;
      int value3;
      int value4;
};

/*******************************************************************************************
Author:  Muharrem Ugur Aksu
         Naval Postgraduate School
         Computer Science Department
Date: 20 July 2006
File Name: intmsg5.msg
*******************************************************************************************/

message IntMsg5
{
   fields:
   int value1;
   int value2;
   int value3;
   int value4;
```

```
   int value5;
};

/*****************************************************************************
Author:  Muharrem Ugur Aksu
         Naval Postgraduate School
         Computer Science Department
Date: 20 July 2006
File Name: intmsg6.msg
*****************************************************************************/

message IntMsg6
{
    fields:
    int value1;
    int value2;
    int value3;
    int value4;
    int value5;
    int value6;
};

/*****************************************************************************
Author:  Muharrem Ugur Aksu
         Naval Postgraduate School
         Computer Science Department
Date: 20 July 2006
File Name: omnetpp.ini
*****************************************************************************/

# contains general settings that apply to all simulation runs
# and all user interfaces.
[General]

# allows for implementing real_time simulation.
scheduler-class = "cRealTimeScheduler"
realtimescheduler-scaling=1000
# this is an instance of the compound module ShuttleSystem.
network = SHUTTLE_SYSTEM
# seed initialization for random number generation (any 32-bit value).
```

```
seed-0-mt=532569
# when enabled, lists the name of ini file entries for which
# the default values were used.
ini-warnings = yes
warnings = yes
# rename statistics output file names.
snapshot-file = ShuttleSystem.sna
output-vector-file = ShuttleSystem.vec
output-scalar-file = ShuttleSystem.sca
# duration of the simulation in simulation time.
sim-time-limit = 1000000s
# duration of the simulation in real time.
//cpu-time-limit= 600s
#specifies the total stack size in kilobytes.
total-stack-kb = 2048 ; 8MByte, increase if necessary

# contains per run settings which
# take precedence over the overall settings.
[Run1]
[Run2]

# contains cmdenv specific settings.
[Cmdenv]

# selects normal or express mode.
express-mode = no
module-messages = yes
//verbose-simulation = yes
display-update = 0.5s

# constains tkenv specific settings.
[Tkenv]
```

```
default-run=1
# write ev output to the tkenv main window.
use-mainwindow = yes
# print banners for each message.
print-banners = yes
slowexec-delay = 300ms
# number of events executed between two display updayes
# when in fast execution mode.
update-freq-fast = 10
update-freq-express = 100
breakpoints-enabled = yes

[DisplayStrings]

# contains values for module parameters that did not
# get a value inside the ned files.
[Parameters]

# configures recording of output vectors.
[OutVectors]
**.enabled = yes

/*****************************************************************************
Author:  Muharrem Ugur Aksu
         Naval Postgraduate School
         Computer Science Department
Date: 20 July 2006
File Name: ShuttleSystem.ned
*****************************************************************************/

simple Shuttle //
    gates:
        out: send_ready;
        out: request_order;
        out: request_next_station;
        out: order_completed;
```

```
        out: send_bid;
        out: route_demand;
        in: order;
        in: next_station;
        in: order_confirmed;
endsimple

simple Manager
    gates:
        in: send_ready;
        in: request_order;
        in: request_next_station;
        in: order_completed;
        in: send_bid;
        in: route_demand;
        out: order;
        out: next_station;
        out: order_confirmed;
endsimple

module PADERBORN_SHUTTLE_SYSTEM

    submodules:

        Shuttles_And_Customers: Shuttle; //
            display: "o=#ff8040,#804000;i=block/users_l,#ff8080;p=112,124";
        Manager: Manager; //
            display: "o=,,4;i=block/browser_l,maroon,10;p=440,124";
    connections:

        Shuttles_And_Customers.send_ready -->
            Manager.send_ready display "o=#808040;m=m,20,0,20,0"; //
        Shuttles_And_Customers.request_order -->
            Manager.request_order display "o=#ffff80;m=m,0,12,0,12"; //
        Shuttles_And_Customers.request_next_station -->
            Manager.request_next_station display "o=#ff0080;m=m,60,60,40,60";

        Shuttles_And_Customers.order_completed -->
            Manager.order_completed display "m=m,36,84,36,84"; //
        Shuttles_And_Customers.send_bid -->
            Manager.send_bid display "o=#00ff80;m=m,8,40,8,40";
        Shuttles_And_Customers.route_demand -->
            Manager.route_demand display "o=#ffffff;m=m,48,100,48,100";
```

```
        Manager.order --> Shuttles_And_Customers.order display "o=#ffff80;m=m,0,26,0,26";
        Manager.next_station -->
            Shuttles_And_Customers.next_station display "o=#ff0080;m=m,8,72,8,72";
        Manager.order_confirmed -->
            Shuttles_And_Customers.order_confirmed display "o=#0080ff;m=m,16,52,16,52";
endmodule
network SHUTTLE_SYSTEM : PADERBORN_SHUTTLE_SYSTEM
endnetwork
```

B. AEG BASED ENVIRONMENT BEHAVIOR MODEL CODE:

The code included in this section defines the behavior of the environment of the SUT with attributed event grammars (AEG). Test drivers were randomly generated from this AEG based environment behavior model.

```
/* Author:  Muharrem Ugur Aksu            */
/*          Naval Postgraduate School     */
/*          Computer Science Department   */
/* Date: 20 July 2006                        */
/* File Name: s4r25.aeg                   */

GLOBAL {
   int transit_fee; /* toll for using the tracks */
   int transit_wear; /* wear incurred for using the tracks */
   int maintenance_wear; /* restored wear value after maintenance */
   int maintenance_fee; /* maintenance fee */
}

RULE Shuttle {
   int start; /* start station of an order */
   int destination; /* final station of an order */
   int shuttle_id; /* unique shuttle identification no */
   int shuttle_at_station; /* current location of a shuttle */
   int capital; /* capital status of a shuttle */
   int wear; /* maintenance status of a shuttle */
   int retired; /* a flag for shuttle bankruptcy */
   int payment; /* money received after order completion */
   int bid; /* bid made by a shuttle for a given order */
   int ord_confirmed; /* a flag for order assignment  */
   int received_order; /* a flag for order offers from broker */
   int distance; /* number of stations for an order */
   int order_request_no; /* auxiliary variable */
```

```
}

RULE Customers {
   int requested_start_station;
   int requested_destination_station;
}

ShuttleSystem :
   /
   transit_fee = get_transit_fee();
   transit_wear = get_transit_wear();
   maintenance_wear = get_maintenance_wear();
   maintenance_fee = get_maintenace_fee();
   /
   {Shuttles, Customers};

Shuttles:
   /***CHANGE NUMBER OF SHUTTLES HERE***/
   {*Shuttle*}(==5);

Customers:
   /
   Customers.requested_start_station = 0;
   Customers.requested_destination_station = 0;
   /
   (* [P(70)/get_random_request(Customers.requested_start_station,
                                Customers.requested_destination_station);
            send_customer_request(Customers.requested_start_station,
                                  Customers.requested_destination_station);/]
   /***CHANGE NUMBER AND FREQUENCY OF CUSTOMER REQUESTS HERE***/
   *)(==1500)(EVERY 10 sec);

Shuttle :
      /
   Shuttle.shuttle_id = unique_id();
   Shuttle.start = 0;
   Shuttle.destination = 0;
   Shuttle.shuttle_at_station = get_shuttle_at_station();
   Shuttle.capital = get_capital();
   Shuttle.wear = maintenance_wear;
   Shuttle.retired = 0;
   Shuttle.payment = 0;
   Shuttle.bid = 0;
```

```
   Shuttle.ord_confirmed = 0;
   Shuttle.received_order = 0;
   Shuttle.distance = -1;
   Shuttle.order_request_no = 0;
   send_ready(Shuttle.shuttle_id);/
(*
   /Shuttle.order_request_no = 0;/
   (*
      /request_order(Shuttle.shuttle_id, Shuttle.order_request_no);/
      wait_order_and_send_bid
      /***WHEN THERE IS ONLY ONE MORE ORDER TO BE OFFERED IN THE QUEUE…***/
      /***REQUEST FOR AN ORDER ONE LAST TIME AND WAIT FOR ORDER CONF.***/
      WHEN (Shuttle.order_request_no == -1)
      (
         /request_order(Shuttle.shuttle_id, Shuttle.order_request_no);
         BREAK; /
      )
      /***WHEN THERE IS NO AVAILABLE ORDER IN THE QUEUE…***/
      /***WAIT ONE ORDER PROCESSING PERIOD OF TIME AND REQUEST AGAIN***/
      WHEN (Shuttle.order_request_no == -2) /BREAK;/
   /***MAKE SURE THIS NUMBER IS EQUAL TO NUMBER OF SHUTTLES***/
   *)(==5)
   wait_order_confirmation
   WHEN (Shuttle.ord_confirmed)
   (
      /Shuttle.payment = Shuttle.bid;/
      (*
         WHEN (ENCLOSING Shuttle.shuttle_at_station != ENCLOSING Shuttle.start)
         (
            /move_to_start_station(Shuttle.shuttle_id, Shuttle.shuttle_at_station);/
            wait_next_station
            process_move
         )
         ELSE /BREAK;/
      /***THIS IS THE MAX DISTANCE BETWEEN TWO FARMOST STATIONS***/
      *)(==5)
      (*
         WHEN (ENCLOSING Shuttle.shuttle_at_station != ENCLOSING Shuttle.destination)
         (
            /request_next_station(Shuttle.shuttle_id, Shuttle.shuttle_at_station);/
            wait_next_station
            process_move
         )
```

```
         ELSE /BREAK;/
      /***THIS IS THE MAX DISTANCE BETWEEN TWO FARMOST STATIONS***/
      *)(==5)
      process_order_completion
   )
/***MAIN LOOP - INCREASE TO GENERATE MORE DATA***/
*)(==50);

wait_order_and_send_bid:
   (* CATCH order(ENCLOSING Shuttle.shuttle_id,
                  ENCLOSING Shuttle.start,
                  ENCLOSING Shuttle.destination,
                  ENCLOSING Shuttle.distance,
                  ENCLOSING Shuttle.order_request_no)
      /ENCLOSING Shuttle.received_order = 1;/
      calculate_and_send_bid
      END_CATCH
   *)(==2)(EVERY 5 sec);

calculate_and_send_bid:
   WHEN(ENCLOSING Shuttle.received_order)
   (
      /ENCLOSING Shuttle.bid = calculate_bid(ENCLOSING Shuttle.distance);/
      WHEN (ENCLOSING Shuttle.order_request_no != -2)
         /send_bid(ENCLOSING Shuttle.shuttle_id, ENCLOSING Shuttle.bid,
                   ENCLOSING Shuttle.start, ENCLOSING Shuttle.destination);/
      /ENCLOSING Shuttle.received_order = 0;/
   );

wait_order_confirmation:
   (* CATCH order_confirmed(ENCLOSING Shuttle.shuttle_id,
                            ENCLOSING Shuttle.ord_confirmed,
                            ENCLOSING Shuttle.start,
                            ENCLOSING Shuttle.destination,
                            ENCLOSING Shuttle.bid)
      END_CATCH
      *)(==2)(EVERY 5 sec);

wait_next_station:
   (* CATCH next_station(ENCLOSING Shuttle.shuttle_id,
                         ENCLOSING Shuttle.shuttle_at_station)
      END_CATCH
   *)(==2)(EVERY 10 sec);
```

```
process_move:
   WHEN (ENCLOSING Shuttle.wear > 0)
   (
      /ENCLOSING Shuttle.capital = ENCLOSING Shuttle.capital - transit_fee;
       ENCLOSING Shuttle.wear = ENCLOSING Shuttle.wear - transit_wear; /
   )
   ELSE
   (
      /ENCLOSING Shuttle.capital =
       ENCLOSING Shuttle.capital – maintenance_fee - transit_fee;
       ENCLOSING Shuttle.wear = maintenance_wear;/
   );

process_order_completion:
   [P(80) order_completed_in_time]
   [P(20) late_order_completion
      /ENCLOSING Shuttle.capital = ENCLOSING Shuttle.capital – get_punishment();/]
   /ENCLOSING Shuttle.capital = ENCLOSING Shuttle.capital + ENCLOSING Shuttle.payment;
   ENCLOSING Shuttle.ord_confirmed = 0;/
   WHEN (ENCLOSING Shuttle.capital <= 0)
   (
      /ENCLOSING Shuttle.retired = 1;/
   )
   /send_order_completed(ENCLOSING Shuttle.shuttle_id,
                         ENCLOSING Shuttle.retired,
                         ENCLOSING Shuttle.capital);/;
```

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Turkish Army Academy Library
   Turkish Army Academy
   Ankara, Turkey

4. Mikhail Auguston
   Naval Postgraduate School
   Monterey, California

5. Man-Tak Shing
   Naval Postgraduate School
   Monterey, California